TAMPERE UNIVERSITY OF TECHNOLOGY

LASSE LEHTONEN

TRANSACTION GENERATOR - TOOL FOR NETWORK-ON-CHIP BENCHMARKING

Master's thesis

Examiners: Prof. Timo D. Hämäläinen and Dr. Tech. Erno Salminen
Examiners and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 5th March 2014.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Engineering
LASSE LEHTONEN: Transaction Generator - Tool for Network-on-Chip Benchmarking
Master of Science Thesis, 50 pages
7th May 2014
Major: Embedded Systems
Examiners: Prof. Timo D. Hämäläinen and Dr. Tech. Erno Salminen
Keywords: Network-on-Chip, NoC, benchmarking, simulation, Transaction Generator

This thesis focuses on benchmarking on-chip communication networks. Multiprocessor System-on-Chips (MP-SoC) utilizing the Network-on-Chip (NoC) paradigm are becoming more prominent. Required communication capabilities differ considerably among the diverse set of application categories. A standard and commonly used benchmarking methodology for networks is needed to ease finding a suitable network topology and its configuration parameters for those applications.

This thesis presents a simulation based tool Transaction Generator (TG) for evaluating NoCs. TG conforms the Open Core Protocol - International Partnership (OCP-IP) NoC benchmarking group's proposed methodology. TG relies on abstract task graphs made after real Multiprocessor System-on-Chip (MP-SoC) applications or synthetic test cases. TG simulates the workload tasks on Processing Elements (PEs) and generates the network traffic accordingly and collects statistics.

TG was initially introduced in 2003 and this thesis presents the current state of the tool and the modifications made. The work for this thesis includes refactoring the whole program from a TCL and C++ SystemC 1 based code generator to a C++ SystemC 2 based dynamic simulation kernel. In addition to the refactoring, new features were implemented, such as memory modeling with the Accurate Dynamic Random Access Memory (DRAM) Model (ADM) package, possibility of simulating Mobile Computing System Lab (MCSL) NoC Traffic Patterns workload models and diversity to modeling the workload.

The current implementation of TG consists of 10k lines of code for the simulator core, the result of this thesis, and 50k lines of code for the support programs and example NoC models. Thesis presents 3 example use cases requiring around 100 simulations, which can be executed and analyzed in a work day with the TG.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietoliikenne-elektroniikan koulutusohjelma
LASSE LEHTONEN: Transaction Generator - Tool for Network-on-Chip Benchmarking
Diplomityö, 50 sivua
7. Toukokuuta 2014
Pääaine: Sulautetut järjestelmät
Tarkastajat: prof. Timo D. Hämäläinen ja TkT Erno Salminen
Avainsanat: piirinsisäinen kytkentäverkko, suorituskykyvertailu, simulointi, NoC

Monen prosessorin järjestelmäpiirit (Multiprocessor System-on-Chip, MP-SoC) käyttävät yhä enenevissä määrin hyväkseen piirinsisäisiä kytkentäverkkoja (Network-on-Chip, NoC) kommunikaationsa välittämiseen. Järjestelmien vaatiman kommunikaation määrällinen ja laadullinen tarve vaihtelee huomattavasti eri sovelluskategorioiden välillä. Standardi ja yleisesti käytetty metodologia kytkentäverkkojen vertailuun tarvitaan helpottamaan tarvittavan kytkentäverkkotopologian ja sen konfiguraation löytämiseksi eri sovelluksille.

Tässä diplomityössä esitellään simulaatioon perustuva työkalu Transaction Generator (TG) piirinsisäisten kytkentäverkkojen suorituskykyvertailuun ja analysointiin. TG noudattaa Open Core Protocol - International Partnership (OCP-IP) NoC benchmarking group:n määrittelemää metodologiaa. TG perustuu abstraktien sovellusmallien simuloitiin, jotka ovat mallinnettu oikeiden sovellusten tai synteettisten testikuvioiden perusteella. Sovellusmallit simuloidaan pelkistetyn laitteistomallin päällä, joka luo työkuorman kytkentäverkolle, josta kerätään statistiikkaa vertailuun.

TG on alunperin esitelty vuonna 2003 ja tässä työssä esitellään sen nykyinen tila ja työssä lisätyt uudet ominaisuudet. Työn anti kattaa TG:n uudelleenimplementoinnin TCL- ja SystemC 1 -kielisestä lähdekoodin generaattorista C++ SystemC 2 kielillä toteutettuksi dynaamiseksi simulaatiokerneliksi. Työkaluun lisättiin muunmuassa tuki DRAM muistimalleille, välimuistin mallinnus, tuki Mobile Computing Systems Lab (MCSL) NoC Traffic Patterns sovellusmallien simuloimiselle ja mahdollisuus simuloitavan laskenta- ja tiedonvälityskuorman monipuolisemmalle kuvaamiselle.

TG:n nykyinen toteutus koostuu 10 000 lähdekoodirivin kernelistä, joka toteutettiin tähän työhön, ja 50 000 rivistä lähdekoodia sen apuohjelmille ja esimerkkiverkoille. Työssä esitellään kolme käyttötapausta ohjelmasta, jotka vaativat noin 100 simulaatiota. Nämä simulaatiot kykenee ajamaan ja analysoimaan yhdessä työpäivässä TG:n avulla.

PREFACE

The work for this M. Sc. thesis was carried out at the Department of Pervasive Computing at Tampere University of Technology as a part of the NOCBENCH project funded by Academy of Finland.

I would like to thank Prof. Timo D. Hämäläinen for the opportunity to work on this project and Dr. Tech. Erno Salminen for the sufficiently long design meetings, when planning the new features, and elaborate guidance with the thesis. I would also like to thank Esko Pekkarinen for the help verifying the program during its development.

I'm grateful for my current co-workers and friends for the healthy competition that gave me an additional motivation to finish the thesis, and my family for the support during the process.

Tampere, 7th May 2014

Lasse Lehtonen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

f_{pe}	Operating frequency of the processing element.
IPC	Instructions per cycle.
$\mathcal{N}(x;\mu,\sigma^2)$	Normal distribution with mean of x or μ, and variance σ^2.
t	Time.
$\mathcal{U}(a,b)$	Uniform distribution between a and b.

ADM	Accurate Dynamic Random Access Memory (DRAM) Model
CPU	Central Processing Unit
CSV	Comma-Separated Values
DMA	Direct Memory Access
DRAM	Dynamic Random-access Memory
DVB-C	Digital Video Broadcasting - Cable
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
FIFO	First In, First Out
HDL	Hardware Description Language
HNOCS	Heterogeneous Network-on-Chip Simulator
IP	Intellectual Property
ISS	Instruction Set Simulator
KPN	Kahn Process Network
LTE	Long-Term Evolution
MCSL	Mobile Computing System Lab
MoC	Model of Computation

MP-SoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OCP-IP	Open Core Protocol - International Partnership
PE	Processing Element
RTL	Register Transfer Level
SoC	System-on-Chip
TCL	Tool Command Language
TG	Transaction Generator
TL	Transaction Level
TLM	Transaction Level Model
VC	Virtual Channel
VHDL	Very High Speed Hardware Description Language
XML	Extensible Markup Language

1. INTRODUCTION

Microchip technology still continues to develop close to Gordon E. Moore's observations known as Moore's law [30]. He observed that the number of transistors on integrated circuits doubles approximately every two years. This increase in capacity allows the creation of increasingly complex designs on a single chip. Nowadays integrating multiple functions on a single chip is common and such a device is known as System-on-Chip (SoC).

Continuous development allows the instantiation of multiple processors, memory elements, interfaces and other functions on a chip, often dubbed as Multiprocessor System-on-Chip (MP-SoC), to raise the overall computing performance by parallel computation [15, 47]. These chips are the foundation of many present day's applications, such as smart phones wich offer great capability to handle multimedia applications which weren't possible in the recent past. The Snapdragon S4 processor [39], for example, is a SoC for mobile applications consisting of a multi-core processor subsystem based on ARM architecture, a modem subsystem for Long-Term Evolution (LTE) technology and a multimedia subsystem among other needed hardware, such as memory.

The rising amount of various internal function blocks and their complex communicational requirements makes the communication between the blocks more difficult. Technologies used on circuit boards, such as point-to-point connections or shared buses, are not applicable for modern SoCs [4], due to multiple problems that make the development and verification more demanding [43]. For examle, the difficulty of synchronization with a single clock source on large designs and the high power consumption of long wires.

Separating the computation from communication allows them to be handled separately in the development and verification processes [20]. New communication networks and network topologies have been designed and studied. The interconnection network is commonly called a Network-on-Chip (NoC) [11, 7]. The study on NoCs still continues as better approaches are needed to suit the vast amount of different application needs. Figure 1.1 shows a conceptual illustration of a modern homogeneous multi-hop network approach for the interconnection of a MP-SoC. Big boxes represent the functional units and the smaller boxes with a cross the routers in the network, that make the decisions how to forward the data to its destination. Arrows

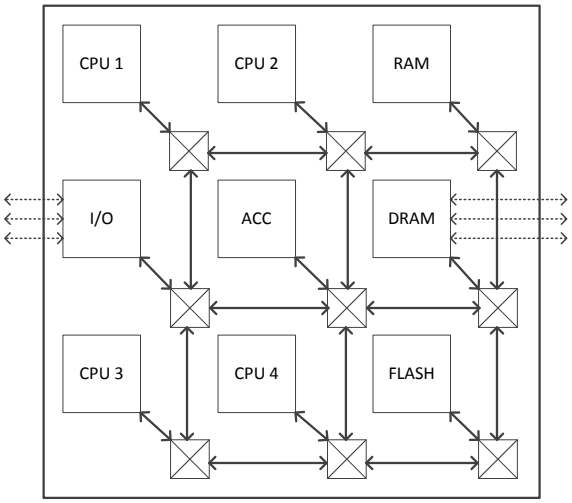

Figure 1.1: Example of a Multiprocessor System-on-Chip with multiple processors (CPU), internal memories (RAM, FLASH), an accelerator (ACC) , controller for external memory (DRAM) and an interface with external hardware (I/O). Each block is connected to a 2-dimensional 3x3 mesh NoC.

between routers mark the communication links that are the physical connections.

This thesis concentrates on a simulation based tool to evaluate and benchmark NoC implementations. Intention of the tool, Transaction Generator (TG), is to help choosing most optimal NoC for certain application or application domain by benchmarking them with common methodology. In practice this means simulation based Design Space Exploration (DSE), in other words finding suitable parameters for the NoCs for different applications.

Initial version of TG was originally created at the Department of Computer Systems at Tampere University of Technology in early 2000s, see [19] for more details of the original version. For this thesis TG was completely rewritten to support modern technologies. The most significant ideas behind the utilized methods are still the same but the internal structure has been changed to support newer SystemC [45] implementation and the SystemC Transaction Level Model (TLM) [9] methodologies instead of the deprecated 1.1 version.

Also many additions were implemented, such as mixed language simulation capability, namely with Hardware Description Languages (HDLs) Very High Speed Hardware Description Language (VHDL) and (System)Verilog, inclusion of Accurate Dynamic Random Access Memory (DRAM) Model (ADM) package [44] for detailed memory simulations, possibility of simulating MCSL NoC Traffic Patterns [26] workload models, enhanced workload modeling, and more precise measurements during simulations.

The resulting program of this thesis has been used, on its various development

stages, in the PhD thesis of E. Salminen [40], in the author's bachelor thesis [23] and the resulting conference paper [25], and in the bachelor thesis of E. Pekkarinen [36] and the resulting conference paper [37].

Thesis is structured as follows. Chapter 2 introduces work related to this thesis, chapter 3 presents the TG generally. Chapters 4 through 8 explains the current implementation details. Chapter 9 demonstrates example use cases and lastly chapter 10 presents the conclusions.

2. RELATED WORK

This chapter presents an introduction to Network-on-Chips (NoCs) and Design Space Exploration (DSE) and NoC simulators and traffic generators related to this thesis.

2.1 Network-on-Chip

NoC is a communication infrastructure for a single integrated circuit based on modular design. The Network-on-Chip paradigm emerged to replace the design-specific communication wiring with a general network to pass data from module to module. Instead of connecting the communicating parties with direct wires, a packet based multi-hop network has been proposed [12]. Multi-hop based networks bring many benefits to the chip design, such as reducing the cross-talk and power dissipation, and allowing the communication wires to be shared between multiple participants.

Modern NoC consists of network interfaces that connect the various Intellectual Property (IP) blocks, the functional units, to the network, routers that switch the data stream according to a decided routing convention and links that connects the routers together and thus creating the network topology [7].

Multiple different topologies have been designed to implement a NoC , such as a mesh, ring and a tree shown in figure 2.1. Usually the choice between them is not obvious to get the best balance between power consumption, operation efficiency, physical area, and other important design aspects.

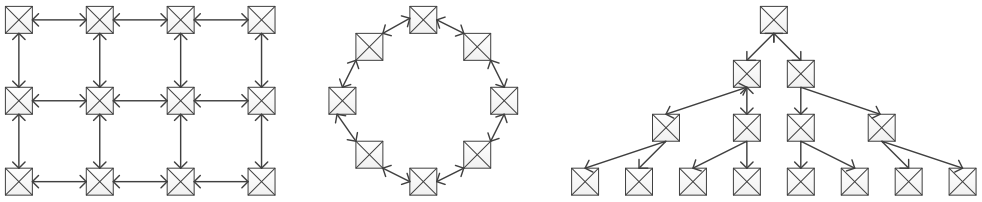

Figure 2.1: Example of different NoC topologies. From left to right: mesh, ring and tree. Image depicts only the routers and the links between them. Functional elements connected to the routers are omitted.

NoC designs often allow multiple ways to customize them, such as to modify the depth of buffers, to adjust routing priorities etc. Moreover, different applications

can be mapped to the hardware in many ways, especially in case of heterogeneous Multiprocessor System-on-Chip. This means deciding for example which processors execute which tasks and how the memory is mapped for these tasks. To find a efficient enough combination for the application's requirements many variations must be tested [13, 42].

2.2 Design Space Exploration

Simulating various mapping combinations to find a suitable solution is called Design Space Exploration (DSE) [43]. DSE is usually done automatically following the diagram in figure 2.2. Functionality is mapped to the hardware architecture and then simulated gathering various performance, resource usage, power consumption, and other metrics that are required to measure its goodness. The metrics are evaluated to see if the configuration meets the requirements for production.

Different heuristic algorithms are used to modify the configuration, for example by changing applications mappings to processors or connecting the hardware IP blocks differently to the NoC.

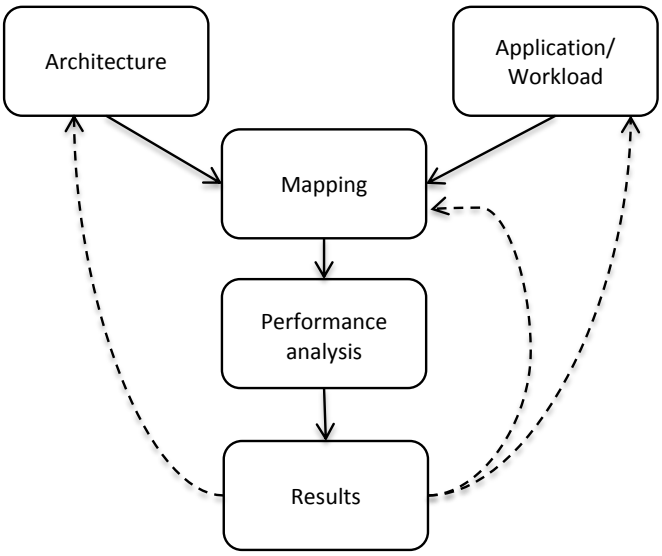

Figure 2.2: Common Design Space Exploration workflow [13]. Continuous arrows mark the order of the main workflow and dashed arrows the optional recursive paths when iterating the design.

The three important aspects of DSE are its accuracy, exploration speed and the amount of work the modeling requires. To gain better exploration speed, it is often required to reduce the timing accuracy, for example by moving from Register Transfer Level (RTL) modeling to Transaction Level Model (TLM). Exploration speed can be increased also by reducing the functional accuracy, for example by moving

from functionally accurate application models to abstract workload models. At the beginning of the application's development more inaccurate models are suitable to coarse-grain elimination of the unsuitable design space. Later more precise models are needed to verify the design choice which often leads to slower exploration speed.

Multiple DSE frameworks have been created for studying and comparing NoCs such as MESH [10], Metropolis [3] and Artemis [38]. MESH [10] is a tool for high-level performance modeling and it is based on layered frequency interleaving. It models the system using a sequencing of logical and physical events. Logical events, in other words the software functionalities, are created from a coarse-grain data set that has been gathered from accurate Instruction Set Simulator (ISS) simulations and interleaved to resources, the computational units, in data-depended manner in faster simulations.

Metropolis is an environment for complex electronic-system design providing support for simulation, formal analysis and hardware synthesis. A metamodel language is used to define the network for example for modeling functionality, architecture and mapping which then can be simulated and verified with tools compatible with the Metropolis environment.

A SystemC based simulation system presented in [21] captures the functionality, timing and interfaces separately allowing co-simulation of multiple abstraction levels. Initial cosimulation of an already existing system is used in [22] to provide abstract traces. Unlike many simulators which use the traces for simulation based performance analysis, [22] uses traces for static analysis of the network performance.

Kahn Process Networks (KPNs) [18] are often used method for describing applications especially for multimedia and other applications that are oriented on streaming the data. They are a Model of Computation (MoC) where applications are modelled as executable processes connected by point-to-point connections separating the computation and communication. Transaction Generator also uses an extended KPN as its MoC.

For example, Artemis project [38] provides a Sesame framework for system level DSE for MP-SoC applications using the KPN MoC taking similar approach as Transaction Generator (TG). Simulation environment is divided to application model, mapping layer and architecture model according to the Y-chart approach shown in figure 2.2.

2.3 NoC Simulators and Traffic Generators

Multiple NoC simulators and traffic generators exist to aid the analysis of the NoCs. This section briefly introduces a portion of the open-source implementations available.

Noxim [32] is a SystemC NoC simulator developed at the University of Catania. It evaluates the throughput, delay and power consumption of the 2D mesh network, which can be customized, for example, by size, buffer depth and routing algorithm, for various customizable traffic patterns.

Booksim [16] is a cycle-accurate interconnection simulator intitially introduced with the book Principles and Practices of Interconnection Networks [11] and it is implemented in C++. Booksim offers multiple NoC topologies, various routing algorithms and allows the user to customize the router architecture. It supports synthetic patterns and previously generated traces for traffic generation and can be integrated with the GEM5 [6] system simulator.

HNOCS (Heterogeneous Network-on-Chip Simulator) [5] is a C++ simulator based on OMNet++ [46] for heterogeneous networks. It supports arbitrary topologies with synchronous, synchronous virtual output queue or asynchronous routers. Traffic can be generated from source to either deterministic or random destinations. Data is sent at randomly distributed or trace file based intervals. HNOCS provides statistical measurements collected by the source, sink and Virtual Channels (VCs) at the network.

TOPAZ [1] is a C++ interconnection network simulator for chip multiprocessors and supercomputers. It supplies multiple network topologies with various configurable router designs. User can, for example, choose from different flow control mechanisms, multicast options, number of virtual channels and pipeline and delay details. Traffic can be generated based on multiple synthetic traffic patterns, such as random, tornado and bit-reversal or from traces.

ATLAS [27] is a NoC generation and evaluation framework written in Java. Framework consists of a NoC generator, traffic generator, and performance and power evaluation. Network can be generated as Very High Speed Hardware Description Language (VHDL) based on configuring parameters, such as the dimension, buffer depth, flow control, number of VCs and routing algorithm. Traffic is generated based on synthetic traffic patters.

3. TRANSACTION GENERATOR

Transaction Generator (TG) is an open-source network traffic generator and benchmarking tool for evaluation and architecture exploration of split-transaction NoCs. Original version of TG has been implemented already in early 2000s [19]. It has also been described in [14, 40] and [41], while the latter two already include some contributions of this thesis. Figure 3.1 illustrates the general idea behind TG's MoC and implementation.

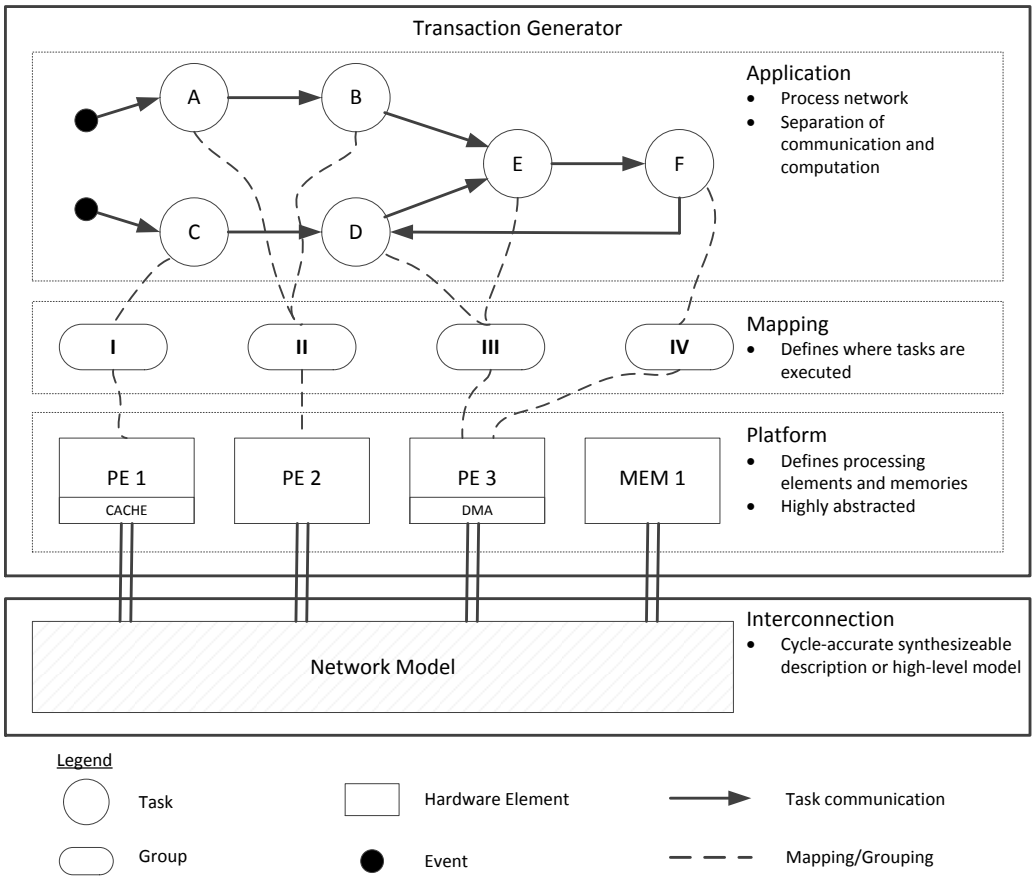

Figure 3.1: Conceptual view of Transaction Generator adapted from [41]. Simulation model is divided into four main parts. Application model defines the behavior of the workload. Mapping assigns tasks to groups and binds them to a particular PE. PEs model the differences between various processors, accelerators and memories. Lastly the communication architecture models the details of the interconnection between the PEs.

TG's current version has been implemented as a part of NOCBENCH project [31], funded by Academy of Finland [2]. The program has been adopted by Open Core Protocol - International Partnership (OCP-IP) Network-on-Chip benchmarking workgroup and is available for download from [33] and [31]. TG is implemented with C++ and SystemC 2 language [45] on a Transaction Level (TL) with a highly abstracted MoC. The MoC allows the workload to be described in variable level of details from simple KPN-style compute-and-fire processes to detailed probabilistic or deterministic descriptions of the computational and communicational behavior.

Transaction Generator includes the Accurate Dynamic Random Access Memory (DRAM) Model (ADM) package [44] for clock cycle accurate memory models for more elaborate simulations. TG is distributed with workload models gathered from literature [37], workload models from MCSL Benchmark Suite[26] and NoC models including synthesizable VHDL, cycle-accurate SystemC and TLM SystemC implementations.

3.1 XML Description

Transaction Generator generates the workload for the NoC based on a description written in Extensible Markup Language (XML) language. This allows the description to be easily generated by program or modified by hand. The Simulation model of is devided into four main parts:

Application model characterises the workload by defining the relationship between computation, communication, and the dependencies for each task.

Mapping allows grouping of tasks and assigning them to the PEs and other resources. It creates a layer of indirection between the application and platform models.

Platform model defines the properties of resources and their connection to the NoC.

Network model is separated from the previous to allow the users to easily integrate their own NoC models to TG.

Clear separation between application model and harware platform model allows easily to modify one of the components separately to quantify their effect on the performance measurements. Separation between communicational and computational requirements enables the workload to describe more accurate behavior depending on which type of processing element the application is mapped and thus executed on. Application model can be used to enforce correct dependencies between communicating tasks to create realistic workload from a real program or to create a synthetic probabilistic traffic patterns.

Figure 3.2 presents a simplified view of the XML source code used to describe the different models.

```
<system>
  <application>
    <task_graph>
      <task id="A"> ...  </task>
      ...
      <task id="F"> ...  </task>
    </task_graph>
  </application>

  <mapping>
    <resource name="PE2" id="0">
      <group id="II">
        <task id="A"/>
      </group>
    </resource>
  </mapping>

  <platform>
    <resource_list>
      <resource id="0" name="PE2" frequency="150" type="RISC" .../>
    </resource_list>
  </platform>

  <constraints>
    <noc class="MESH" type="2x2" frequency="200"/>
    <sim_resolution time="1.0"   unit="fs"/>
    <sim_length     time="100"   unit="ms"/>
    <cost_function  func="..."/>
  </constraints>

</system>
```

Figure 3.2: Model description in pseudo-XML showing the separation of different modeling layers for TG.

In figure 3.2 application's workload model defines tasks from A to F that are used to create the computational workload for PEs and the communicational patterns between tasks. Mapping section binds task A to be run on processing element PE2 and platform section defines the simulation parameters for PE2 such as the operating frequency and its type.

In addition to the application, mapping and platform sections, which are used to model the fixed hardware and software aspects of the simulation, the constraints section allows the user to define the metrics to measure and other settings for individual simulation runs such as the NoC configuration parameters and the simulation's length.

Transaction Generator's XML input allows describing the workload on a very abstract level or with high details and can be easily generated from other workload model descriptions. In addition to the native XML description Transaction Generator also parses and generates traffic from MCSL Benchmark Suite's [26] .rtp and .stp file formats.

3.2 Simulation Results

TG gathers various information during the simulation at configurable intervals and provides summaries of the whole simulation. Information is saved in Comma-Separated Values (CSV) format. Table 3.1 lists the information TG gathers from the simulation. Application category reports the information from the application model and processing element the state of the platform model. Token category defines the information of communication as defined in the application model and packet the information of the smaller data packets the tokens must be split for the network model. Summary provides the results for the whole simulation.

Table 3.1: Summary of statistics TG gathers from the simulation. Snapshot measurements store the information at even intervals during the simulation. Delta measurements report the information averaged or accumulated between the snapshots. Trace measurements are saved at the time they happen and summary provides the measurements from the whole duration of the simulation.

Category	Style	Logged Information
Application	Snapshot	Current time, task name, current state, total times triggered, total amount of bytes sent, current receive buffer usage.
Processing Element	Delta	Current time, PE name, current state, PE utilization, received bytes, sent bytes, receive buffer usage, transfer buffer usage, idle cycles, busy cycles, cycles spent reading, cycles spent sending, cycles spent waiting reception, cycles spent waiting to transfer, and cycles spent for intra-PE communication.
Token	Trace	Token identifier, time when sent, time when received, latency (in absolute time and in receiver's cycles), bytes, number of packets it was split, sender task, receiver task, sender resource, receiver resource, source port, destination port, and type of the transfer.
Packet	Trace	Reception time, token identifier it is part of, size, source port, destination port, and type of the transfer.
Summary	Summary	Cost function results, PE statistics, memory statistics, task statistics, and event statistics.

In addition to the log data mentioned in table 3.1, TG can evaluate simple cost function equations consisting of constants, four basic mathematical operators (x+*/), parenthesis, and various variables from the simulation. Variables include information, such as average utilization of the processing elements, trigger counts

and times, and latency of the communication between tasks.

TG can also connect to Execution Monitor [17], which is a program for real-time monitoring the execution of a System-on-Chip (SoC). Execution Monitor, shown in Figure 3.3, can be used to view the simulation's PE usage and application model statistics during the simulation and afterwards from a recorded trace file.

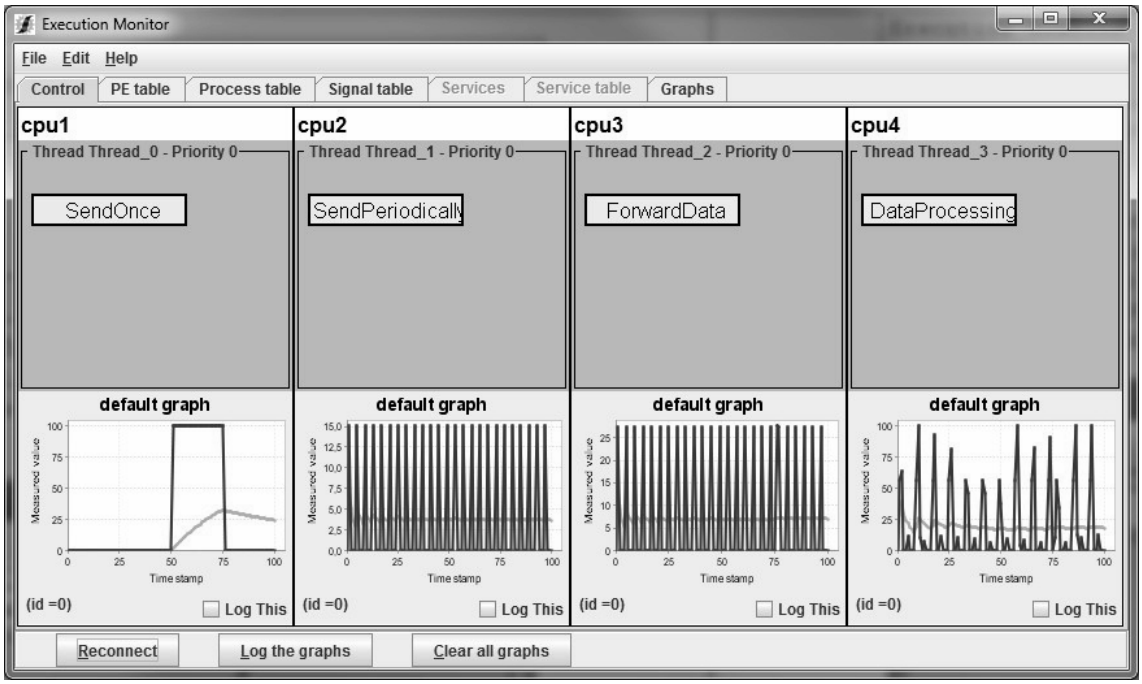

Figure 3.3: Illustration of Execution Monitor [17].

4. APPLICATION MODEL

Transaction Generator's application model can be presented as a set of graphs with unidirectional edges. Vertices on the graph present the computational and communication workload and edges mark the communicational dependencies. Depending on the modeled application there may be anything from a single unidirectional graph to multiple disjoint circular graphs.

Figure 4.1 gives an example of the visualized task graph. Example models the task graph for the channel equalizer application presented in [29]. Example has 3 events, shown as black dots, that are used to start the application by inserting data tokens during the simulation either once, multiple times, or periodically. Named circles represent tasks describing the computational elements of the application, for example adc, load and norm. Edges on the graph represent the data flow with the sizes of data tokens in byte sent from task to task in the direction of the arrow.

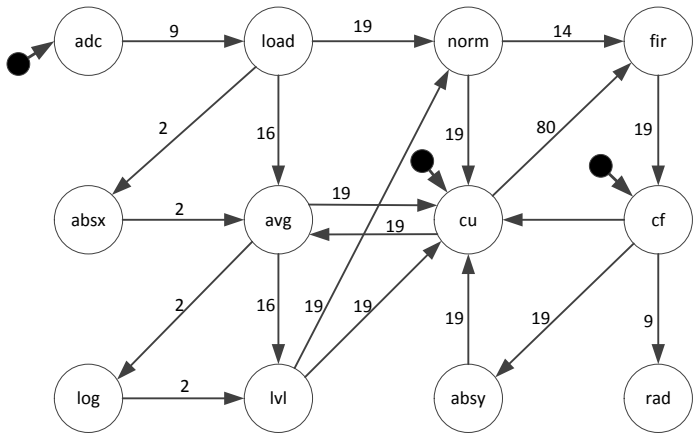

Figure 4.1: Visualized task graph of a channel equalizer application presented in [29]. Small filled dots correspond to TG's events, larger circles with names represent the tasks, and the edges the communicational dependencies and size of the transmissions.

TG's modeling methodology allows to describe the data dependencies needed to trigger the depended tasks, the computational characteristics and communication's destinations and magnitudes in both deterministic and probabilistic ways. All of the workload models may also depend on the task's internal execution history, for example on how many times it has been executed, and the amount of data the task

received. The application model's description consists of:

Events which are used to model the external inputs to the system or the internal timing events to trigger the task graph execution.

Tasks which model the computation and it's relation to the communication with other tasks and memory elements. Computation time depends on the task's internal operation counts and the properties of the PE it's mapped on.

Memory areas that represent the contiguous data memory regions that tasks use from memory elements external to their own PE.

Port connections that bind the tasks together and mark where the tasks are sending data. There may be multiple connections between two tasks to differentiate between data streams which require different response from the recipient. Separate connection information also allows to redirect transmission without needing to modify the individual task descriptions.

Figure 4.2 shows an example of an application description for TG.

```
<application>
  <task_graph>
    <event_list>
      <event name="input1" id="0"/>
      .
      .
    </event_list>
    <task name="A" id="0" class="general">
      .
      .
    </task>
    <task name="B" id="1" class="general">
      .
      .
    </task>
    <mem_area name="mem_a" id="10" size="16386" class="general">
      .
      .
    </mem_area>
    <port_connection src="6" dst="202"/>
      .
      .
    <port_connection src="103" dst="204"/>
  </task_graph>
</application>
```

Figure 4.2: Example of an application description for TG.

Example of the possible successive tags inside a application tag, presented in figure 4.2, describes a task graph containing events, tasks, memory areas, and port

connections. Elements are contained in a task graph tag, which is just for clarity by grouping tags that are related to each other, as there might be multiple task graphs in a one application model description.

Next sections explain the details in Transaction generator's application model. More details about how to describe them in XML can be found from the Transaction Generator's technical documentation [24].

4.1 Event

Events are the stimuli to the application model. At least one event is required to trigger and start the execution of the application model in the first place. Events do not describe any specific physical entity in the simulation model but can be used to model a behavior that is not part of TG MoC. Events may, for example, be used to model some external input to the system, such as the raw signal from a Digital Video Broadcasting - Cable (DVB-C) connection to a task graph modeling a set-top box, or an internal timer for tasks that generate data at certain intervals.

Events can be used to trigger multiple unconnected tasks, for example, to create synthetic traffic patterns, such as the commonly used uniform distribution or bit complement patterns. Events don't utilize network model for sending their payload and their payload doesn't take any time to receive for the task models, even though otherwise they behave similarly to a token send by other task. Event behavior is described with following parameters.

Destination is the outgoing port index for connecting it to tasks.

Data amount defines how many bytes are sent to the receiving tasks.

Frequency describes how often does the event send data.

Offset is the time from the beginning of the simulation when this event is evaluated for the first time.

Evaluation count defines how many times event happens in the simulation which can be a discrete amount or indefinitely.

Probability defines the chance for event to send data when the event is evaluated.

Example XML description in figure 4.3 defines two events for the system named `input1` and `timer1`. Event `input1` is only executed once and it will happen at time 0.03 seconds after the beginning of the simulation. On the other hand event named `timer1` will be executed every 0.02 seconds starting from when the simulation has progressed 0.1 seconds.

```
<event_list>
  <event id="0" out_port_id="1" name="input1" count="1"
         amount="1" offset="0.03" prob="1"/>
  <event id="1" out_port_id="2" name="timer1"
         amount="1" offset="0.1" period="0.02" prob="1"/>
</event_list>
```

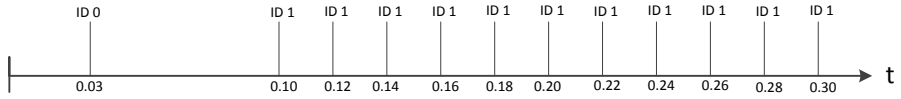

Figure 4.3: Example of describing two events. Event named `input1` is executed once at 0.03 seconds from the beginning of the simulation. Event `timer1` is executed repeatedly every 0.02 seconds starting from 0.1 seconds.

4.2 Task

Tasks are the most important part of the application model. Their descriptions contain the information needed to create the communicational and computational workload to mimic real applications or to generate synthetic models with either deterministic or probabilistic behaviors for the network load.

Description lists the unidirectional input and output ports used for communication with other tasks and memory models and the triggering conditions which describe the behavior after task has received tokens to either one or multiple input ports. The input and output ports are connected together through port connections.

Port connections are one of TG's ways to handle separation between task models. Port connection section of the XML source description for TG allows all of task communication to be redirected without modifying the description of the sender task or the receiving task descriptions. Each of the port identification number defined for the tasks and events must be used only in one port connection.

Figure 4.4 shows an example description and illustration of a task graph with three tasks and how their input and output ports are connected together with port connections.

4.2.1 Trigger

Task's response to received data packets or tokens is controlled by one or multiple triggering conditions. A data token reception may activate multiple triggers. Activated triggers are processed to create a list of computation and communication orders for the task to execute.

There are two kind of dependency handling for triggers receiving data tokens from multiple sources. OR-type trigger's condition is satisfied when it receives a

```
<task_graph>
  <task name="task1" ...>
    <out_port id="1"/>
    <out_port id="2"/>
    <trigger> .. </trigger>
      .
      .
  </task>
  <task name="task2" ...>
    <in_port id="3"/>
    <out_port id="4"/>
    <trigger> .. </trigger>
      .
      .
  </task>
  <task name="task3" ...>
    <in_port id="5"/>
    <in_port id="6"/>
    <trigger> .. </trigger>
    <trigger> .. </trigger>
      .
      .
  </task>
  <port_connection src="1" dst="3"/>
  <port_connection src="2" dst="5"/>
  <port_connection src="4" dst="6"/>
</task_graph>
```

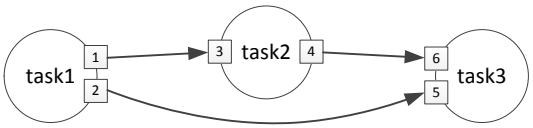

Figure 4.4: Example of a description of a task graph with three tasks and their port connections. Numbered squares on the tasks present the input and output ports and their identifiers used for port connections.

data packet to one of its input ports and AND-type trigger's condition is satisfied only when all of its input ports have received a data token. This allows more diverse options for modeling the task's behavior than is possible with only traditional KPN models. This, for example, allows creating tasks that only execute after receiving all the needed information from multiple sources or immediately after receiving a token regardless of the sender.

All triggers may also depend on the execution history of the task. This allows the task's behavior to chance over time for the same input ports which is necessary for more detailed application models. The behavior can even be different for every time a task is executed allowing the model to be used for storing a trace gathered from real system simulations. Figure 4.5 shows an example of a task's trigger.

Example in figure 4.5 depicts a trigger that is executed after it has received a data token to both its input ports as it's a AND-type trigger. Each trigger keeps a

```
<trigger dependence_type="and">
  <in_port id="5"/>
  <in_port id="6"/>
  <exec_count>
    .
    .
  </exec_count>
  <exec_count max="0">
    .
    .
  </exec_count>
  <exec_count mod_phase="0" mod_period="2">
    .
    .
  </exec_count>
  <exec_count max="8">
    .
    .
  </exec_count>
</trigger>
```

Figure 4.5: Example of a trigger description.

internal count of the times it has been triggered during the simulation. Execution count tags are used to define the behavior of the task with regard to its execution history, that is the trigger's counter. Execution counts can be defined to limit their execution to happen only once in the trigger's execution history, multiple times or to happen periodically. They are defined with following parameters:

mod_period defines a limit for the current execution count. When the trigger's count reaches this limit it is reset back to zero. This acts as a modulus for the trigger's count allowing the modeling of periodically executed statements. If this parameter is omitted the trigger's counter will not be reset during the simulation and will reflect the actual amount of how many times the trigger has been fired during simulation.

mod_phase is used to limit the execution of the current execution count to only when the trigger's count is a certain number. If `mod_period` is defined the current execution count is executed periodically. If it is not defined then the execution count is only executed once during the simulation.

min limits the execution to only those times when the trigger's execution counter is at least a certain value.

max similarly limits the execution to a certain range of trigger's execution counter's range by defining the maximum value it can have to execute the current execution count.

For example the execution counts shown in the example of figure 4.5 define four different behaviors.

First execution count is always executed, by not defining any restricting parameter for it. Second execution count is only executed on the first time the trigger is fired. Third execution count defines operations for the task that are executed every other time the trigger fires. Lastly the fourth one defines a execution count for operations to be executed only during the eight first times the trigger has been triggered.

4.2.2 Computation and Communication

Once trigger's execution condition has been satisfied it's list of computation and communication statements are evaluated. To calculate the amount of clock cycles to execute or the the amount of bytes to send to or read from other tasks or memories Transaction Generator's XML input format allows the use of polynomial and distributional equations. These equations may be static or depend on the amount of bytes received. Three choices are available. Polynomial equation 4.1 depends on the amount of received data x except for the constant term a_0.

$$a_n x^n \ \cdots \ a_2 x^2 + a_1 x + a_0 \tag{4.1}$$

First choice of distributions is the uniform distribution 4.2. This is the only one that doesn't depend on the amount of incoming data but provides a uniform random amount which is useful when only the range of amount is known but cannot be related to the execution history or the amount of data received.

$$\mathcal{U}(a, b) \tag{4.2}$$

Second option for distributional amount is the normal or Gaussian distribution. The mean parameter can be either a constant μ or the amount of bytes received x and the variance or standard deviation parameter σ^2 is constant.

$$\mathcal{N}(x; \mu, \sigma^2) \tag{4.3}$$

All of the equations can be combined to calculate the final amount of instructions to execute or the bytes to send or read. Each of the computation and communication statements are also dependent on a probability to be executed. If the probability is less than 1.0 (100%) the behavior of that part of task becomes stochastic. The probability is checked every time the trigger is evaluated. This allows for example to model a behavior where task's execution from time to time takes a branch which needs more calculation than other branches. Figure 4.6 illustrates the insides of exe-

cution count tags defining how the amounts of computational and communacational are described.

```
<exec_count>
  <op_count>
    <int_ops>
      <polynomial>
        <param value="20" exp="0"/>
        <param value="1"  exp="1"/>
      </polynomial>
    </int_ops>
    <float_ops>
      <distribution>
        <normal mean="100" standard_deviantion="15"/>
      </distribution>
    </float_ops>
    <mem_ops>
      <distribution>
        <uniform min="30" max="60"/>
      </distribution>
    </mem_ops>
  </op_count>
</exec_count>
```

Figure 4.6: Example of a workload calculation for amounts of integer, floating point and memory operations to perform.

As seen in the example XML description of figure 4.6 the computation statements are divided in three groups which allow to differentiate between integer, floating point and memory operation instructions. The example defines a 20 plus the amount of bytes received integer operations, floating-point operations randomly selected from uniform distribution with mean of 100 and standard deviation of 15, and memory operations randomly selected between 30 and 60 to be executed. The actual amount of clock cycles are then calculated based on the charasteristics of the PE the task is running on.

Communication statements are divided in two groups - send statements and read statements. Send statements are used to send data packets to other tasks or to write to memory models. Read statements are only used to fetching data from the ADM memory models. All communication statements are attached with output port information that directs the packet to correct recipient and a probability of taking place. Figure 4.7 presents the tags used to send data tokens between tasks and to write and read data from the memory models.

In the example shown in the figure 4.7 the first send tag presents normal send of 100 bytes from a task to another. Sends from one task to another are defined with following parameters:

out_id which refers to the corresponding output port listed at the beginning of a

```
<exec_count>
  <send out_id="0" prob="0.5">
    <byte_amount>
      <polynomial>
        <param value="100" exp="0"/>
      </polynomial>
    </byte_amount>
  </send>
  <send out_id="1" prob="1">
    <byte_amount>
      <polynomial>
        <param value="200" exp="0"/>
      </polynomial>
    </byte_amount>
    <burst_length>
      <distribution>
        <uniform min="4" max="16"/>
      </distribution>
    </burst_length>
  </send>
  <read out_id="2" resp_id="3" prob="1">
    <byte_amount>
      <polynomial>
        <param value="32" exp="0"/>
      </polynomial>
    </byte_amount>
    <burst_length>
      <distribution>
        <uniform min="4" max="16"/>
      </distribution>
    </burst_length>
  </read>
</exec_count>
```

Figure 4.7: Example of a workload calculation for amounts of data to send or read.

trigger description.

prob that defines the probability of this send taking place when the trigger has been fired.

byte_amount that defines the amount of bytes sent with the same equations as for the execution amounts.

The latter send tag and the read tag in the figure 4.7 show the syntax for a write of 200 bytes to memory and a read of 32 bytes from a memory model. Those tags can additionally also contain a

burst_length tag that is used to define the amount of bytes that are written to or read from consecutive memory addresses. For example, a bigger 1024-byte write to the memory can be split to multiple smaller 32-byte bursts, that might

not be written consecutive addresses affecting the timings of the DRAM and thus performance.

The burst length can also be expressed as a random number. It will be generated only once for the current evaluation of the read or sent statement and generated again at the next time the trigger containing the send fires. All of the bursts of a single read or send token are written to or read from a random memory address that might for example trigger a row change in a DRAM model and thus affecting the completion time of the operation.

Read tags need also an additional response identification attribute to specify the target port for the token sent by the memory element as the read response. This information is described in the task reading to reduce the number of ports needed in the memory model. Memory models thus can use a single output port to send their read responses to multiple targets.

4.3 Memory Area

Memory areas, a new feature of TG implemented for this thesis, model the contiguous regions of data memory that resides outside of the task's PE that is using it. Memory areas are modelled separately to allow mapping them easily to different memory elements. Like memory elements, memory areas are only used when using the ADM to model the memories. Each memory area is defined by the following set of attributes:

Size describes the amount of bytes needed for this memory area.

Input ports that determine the incoming connections for memory writes and read requests.

Output ports to act as a source port for the responses corresponding to read requests.

Figure 4.8 shows a example of a memory area definition.

4.4 MCSL Traffic Patterns

TG, in addition to its own XML description, can also execute and convert Mobile Computing System Lab (MCSL) NoC Traffic Patterns workload models [26]. This feature was implemented for this thesis. The traffic patters include multiple models based on real applications, such as Reed-Solomon code encoder, H.264 video decoder and random sparse matrix solver for electronic circuit simulations. Traffic patterns have been mapped for mesh, torus and fat-tree NoCs of various sizes. The applications have both recorded and statistical model descriptions.

```
<task_graph>
  <mem_area name="mem_area1" id="1" size="1024" class="general">
    <in_port id="2"/>
    <in_port id="4"/>
    <out_port id="7"/>
  </mem_area>
</task_graph>
```

Figure 4.8: Example of a 1024-byte memory area definition that has two input port and an output port.

Example in figure 4.9 shows the format for a recorded application model recorded from a real H.264 video decoder application. Example is mapped to 16 PEs, has 51 tasks connected by 71 edges containing 10 iterations of the original application.

```
1 16 51 71 10
0 0 0 1 3 6 10 15 21 28 36 44 4343 4744 4385 4802 4286 4214 ...
1 0 2 4 7 11 16 22 29 37 45 52 2865 2460 2438 2521 2369 2591 ...
.
.
.
49 10 4 8 12 16 20 24 28 32 36 38 3530 3736 3360 3589 3212 3533 ...
50 11 2 5 8 11 14 17 20 23 26 29 2258 2687 2359 2435 2681 2628 ...

0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 220 217 206 ...
1 1 2 0 0 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 210 216 206 ...
.
.
.
```

Figure 4.9: Snippets from MCSL recorded traffic pattern file. The task execution information after the first line contains 22 parameters in this case (task id, PE id, 10 sequence numbers for the scheduler, 10 execution times). For example, task 1 is mapped to PE0 and executes 2865 operations when it is run the first time. Similarly, the communication edge 1 goes from task 1 to task 2 and specifies 210 bytes to be sent on the first time, 216 on the second and so on.

After the first line that describes the general task and PE information the file format lists a task execution information and after that the task communication information. Execution information lists the tasks, their mapping to PE, a list of sequence numbers for the sequence scheduler, and lastly the recorded execution times. The task communication information lists the source and destination task identification and the message sizes.

TG parses the statistical and recorded formats internally converting them to its native format, which can be simulated as it is or exported to a file.

4.5 Mapping

TG has a separate mapping section to allow the separation of the application model from the resources of the platform model. It defines on which PE the tasks are executed and the memory where memory areas are placed. Tasks can be placed within software platforms and grouped together, which might affect their communication costs. Figure 4.10 shows an example task and memory area mapping to the resources.

```
<mapping>
  <resource      name="cpu0"         id="0" contents="mutable">
    <sw_platform position="movable" id="0" contents="mutable">
      <group     position="movable" id="0" contents="mutable"
                 name="group0">

        <task position="movable" id="0" name="Task0"/>
        <task position="movable" id="1" name="Task1"/>
        <task position="movable" id="2" name="Task2"/>

      </group>
      <group     position="movable" id="1" contents="immutable"
                 name="group1">

        <task position="immovable" id="3" name="Task3"/>

      </group>
    </sw_platform>
  </resource>

  <resource name="Memory1"           id="1" contents="immutable">

    <mem_area position="immovable" id="4" name="MemoryArea4"/>

  </resource>

  .
  .
  .
</mapping>
```

Figure 4.10: Example of the mapping section gluing application model to the resources on the platform model.

In addition to defining the placement of tasks and memory areas for TG, description also provides mapping constraint information for external DSE tools. Resources, that are the PEs and memories, contain a `contents` tag which describes whether their mapping is allowed to be modified by the DSE tool at all. Software platforms and groups have `contents` and `position` tags. Tag `contents` again describes is mapping their contents allowed and the `position` tag defines is the whole group or software platform allowed to be remapped as a whole. The `position` tag for tasks and memory areas define whether it can be remapped by the DSE tool.

5. PROCESSING ELEMENT MODEL

Processing Element models are used to model processors, hardware accelerators, simple memories and other such hardware resources where application tasks can be mapped to. Model is highly abstracted to gain better performance during simulations. Processing Element are characterized by simple parameters for performance and physical aspects. These definitions are separated to a different XML file, called PE library, containing only descriptions of possible PE types. All PE's have following parameters to define them:

Type divides PE into groups such as general purpose processors, memories and different hardware accelerators.

Frequency bounds limit the operating frequency of this type of PEs.

Direct Memory Access (DMA) controller defines whether or not the PE can handle communication and computation at the same time.

Communication overhead stating the timing modifiers relating to communication transactions.

Computation performance list the performance factors for integer, floating point and memory instructions.

Area that is used in DSE optimizations to estimate the cost of the platform in mm^2 or kilogates.

Power consumption for power consumption estimations in DSE optimizations.

Type parameter distinguishes what kind of applications the PE can execute for automatic mapping processes during DSE. For example it can allow for a given application to map tasks representing mathematical functions to accelerator models or general purpose processor models while denying other tasks from being mapped to accelerator that couldn't execute them in the real world.

Operating frequency, inclusion of a DMA controller, computational characteristics and communicational overheads affect the computational and communication operating speeds of the PE to allow simple and effective method for modeling real IPs.

Area and power consumption are used to estimate the cost of the systems in relation to other simulation results during DSE. For example a powerful IP with high energy consumption may achieve lower total power consumption due to shorter execution time when compared to a slower low-power IP.

In the main XML file containing the description of the platform to be simulated the resources modeling instantiated PEs are defined with the following attributes.

Frequency defines the operating frequency of the PE in MHz.

Type refers to an entry in the PE library containing the generic attributes for the PE.

RX buffer size specifies the maximum size of bytes allocated for received tokens that have not been consumed by the task models. Received tokens are considered consumed when their receiving task has read them from in input buffer either by task itself using up PE's processing time or by the DMA unit. After having been consumed the equivalent amount of tokens size is freed from the RX buffer. If the receive buffer becomes full it stalls the reading from network model causing congestion.

TX buffer size defines the size of the buffer for outgoing tokens for the PE. If the transmission buffer ever becomes full it will stall the execution of sending tokens and thus stalls the progression of tasks.

Packet size determines the maximum size of a packet going to the NoC model as many interconnection networks can't handle unbounded streams of data. This removes the need to change the token sizes in the application workload model to suit the network model. If the task defines a token to be sent that is bigger than the maximum packet size, the token will be automatically split into multiple smaller packets. These original tokens are automatically recombined from the smaller packets at the receiving PE or memory model.

Scheduler parameter is used to select a scheduling algorithm for the PE to use when selecting the next task to be executed.

Figure 5.1 shows an example of a processing element description defining a PE. The resource described in the example of figure 5.1 defines a PE that models a Central Processing Unit (CPU) of generic type that operates at 80 MHz frequency.

5.1 Scheduling

PE's scheduling policy decides the order of execution of tasks. TG supports First In, First Out (FIFO), fixed priority and sequence scheduling schemes. Sequence

```
<platform>
  <resource id="0" name="PE2" frequency="80" type="RISC_CPU"
            rx_buffer_size="262144" tx_buffer_size="1024"
            packet_size="16" scheduler="fifo">
    <cache>
      <i_miss line_size="64" mem_area_id="2">
        .
        .
      </i_miss>
      <d_miss line_size="64" mem_area_id="3">
        .
        .
      </d_miss>
    </cache>
  </resource>
</platform>
```

Figure 5.1: Example of a processing element definition.

scheduling allows designer to use a predetermined order of task execution where the PE is forced to wait the data tokens for next task even though there would be other tasks ready for execution.

Application's state in TG is represented as a state machine shown in figure 5.2. All tasks start from WAIT state.

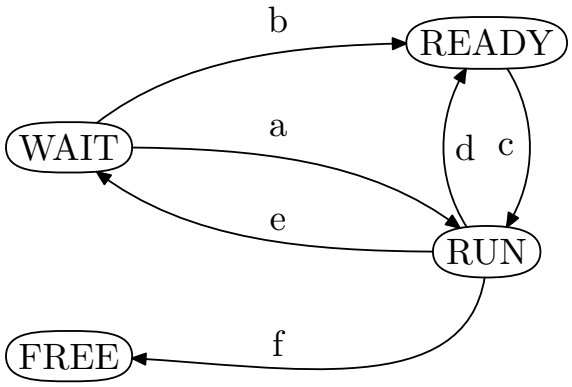

Figure 5.2: Possible states for application model.

Possible state transitions are:

a) Task has previously executed a READ statement and was waiting for a reply from external memory when the reply arrives.

b) Task receives all data tokens to fulfill trigger's condition and moves to READY pool waiting for scheduling.

c) Scheduling algorithm chooses the task for execution.

d) Task still has trigger's to execute but the scheduling algorithm places the task to READY pool.

e) Task has executed its trigger and has no more pending triggers.

f) Task has executed its last trigger and is no longer depended on any data during simulation.

5.2 Execution Model

When task's trigger has fulfilled its dependencies by receiving enough data tokens and the scheduling algorithm changes it to RUN state the simulation engine calculates a list of operations to perform based on the incoming data token sizes and probabilities listed in trigger's XML description. Simulation engine then calculates the data dependent and random amounts and starts to execute the list in order.

Execution speed in time is defined by the operating frequency of the PE, the type of the PE and the availability of a DMA controller. Each type of PE is characterized by how many cycles it takes to execute an instruction in 3 classes of operations, namely integer, floating-point and memory operations. Integer operations are used for the basic instructions that are executed mostly in constant time. Floating-point operations are separated as not all processors have dedicated hardware for them and have to use software emulation which might be significantly slower. Memory operations are meant to describe accesses to local memories not connected through the NoC as these might again depend greatly on the PE they are executed on.

This separation allows to create the application models independent of processor type. For example, figure 5.3 illustrates a situation between `cpu A` with a dedicated floating-point unit and `cpu B` without it.

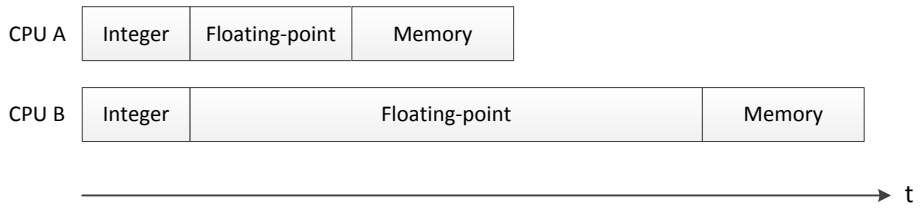

Figure 5.3: Figure illustrates the execution time taken on two different PE model when simulating the same task that includes integer, floating-point and memory operations. In the example the only difference between `CPU A` and `CPU B` is that the `CPU B` models a processing unit without a dedicated floating-point unit.

Each time task's simulation comes to an execution statements it calculates the required cycles needed to execute all consecutive statements:

$$N_{cycles,i,pe} = \frac{N_{int,i}}{IPC_{int,pe}} + \frac{N_{float,i}}{IPC_{float,pe}} + \frac{N_{mem,i}}{IPC_{mem,pe}}, \tag{5.1}$$

where $N_{,i}$ are the integer, floating-point and memory operations calculated for task i and $IPC_{,pe}$ are the PE's instructions per second factors for the same operations. With PE's operating frequency f_{pe} the time to execute these operations is calculated as follows:

$$t_{i,pe} = \frac{N_{cycles,i,pe}}{f_{pe}}. \tag{5.2}$$

TG can model hardware resources operating at different operating frequencies in a simple manner, which in the real world devices creates many difficulties to synchronize with different clock frequency domains.

5.3 Communication Model

In TG's MoC task may send or read data tokens at any point of its execution. This makes it different than KPN model where tasks first consume time for computation and then sends data tokens and are finished. TG's model allows executions, sends and reads to be interleaved in any desired way.

One parameter that affects the PE's performance is the inclusion of a DMA device. In the PE model if it doesn't include a DMA unit it has to stop the computation while it is sending or receiving data from the network. If the PE has a DMA unit it can continue computation while sending. With the PE model with a DMA unit the reception also doesn't take any time provided that the data being read has been received by the PE. Without a DMA unit the reading of a data token also takes time from the PE.

Figure 5.4 illustrates the affect on execution time of the difference between having a DMA unit on the model and not having one.

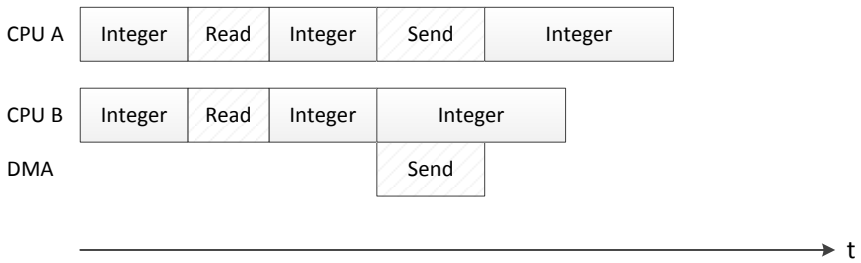

Figure 5.4: Example of how the inclusion of a DMA affects the processing time. The same program is executed on two similar PEs but the `CPU B` has an additional DMA unit.

In the example of figure 5.4 the same program having integer operations and read

and write operations is executed on two similarly powerful PEs. The only difference is that the `CPU B` has an additional DMA unit. The DMA unit doesn't affect the read operation in the example as the task can't continue its execution before the token read has arrived. On the other hand the send operation doesn't block the execution of the later operation and is happening in parallel with the computation.

The other parameters affecting the time communication takes are the PE's communication overhead properties. In the mapping tasks can be assigned to a separate groups and the tasks can reside on different PEs. The communication can be modeled to take different amounts of time depending whether the communicating tasks reside in the same group or not on the same PE or on different PEs. The overheads can be defined to be constant delay for each transaction or depend on the amount of data being sent. Figure 5.5 presents an identical send operation being executed on three different PEs.

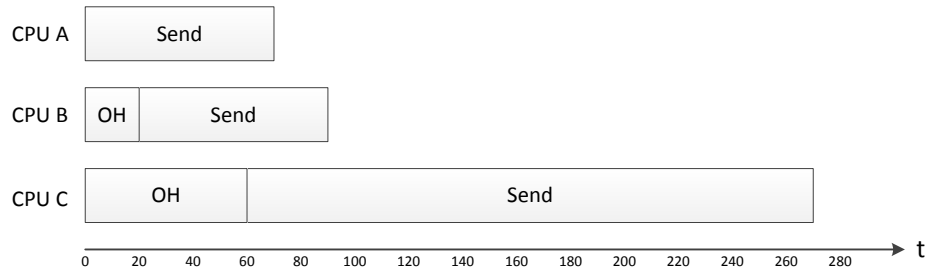

Figure 5.5: Example of how the communication overheads affect the time communication takes. Blocks named OH represent the overhead and Send blocks mark the time for the actual data takes time to be transferred.

In the example shown in figure 5.5 the `CPU A` doesn't have any overheads defined for the transaction it needs. It sends 70 words and executes 1 word/cycle. The PE `CPU B` has a constant 20 cycle initialization overhead for every send operation and executes 1 word/cycle. The PE `CPU C` in addition to the constant 60 cycle overhead also executes the sends three times slower than the other two, taking 60 + (3*70) = 270 cycles.

5.4 Cache Model

Transaction Generator has a simple model for describing cache misses for both the instruction and data memory. Cache model was implemented for this thesis. Only cache read fails are modeled as cache write misses don't have as significant impact on the execution. Cache misses occur when data requested by the application is not in the cache and they are a property of particular piece of code. In TG cache is simplified to be a property of the PE and doesn't vary between tasks on the same

PE. The model allows to define cache miss rates for both the instruction and data memory.

Miss rates are specified using the same equations as with the computation and communication amounts. Result from the equations is interpreted as a clock cycle count to the next cache miss. The cycle count is reduced when the PE model is executing computation and not during reads or writes. When the cycle count reduces to zero the PE model will halt the execution and issue a memory read to the associated memory area. Execution is continued after the read request has been fulfilled. Figure 5.6 shows an example of the XML description for cache misses.

```
<platform>
  <resource id="0" name="PE2" frequency="80" type="RISC_CPU"
            rx_buffer_size="262144" tx_buffer_size="1024"
            packet_size="16">
    <cache>
      <i_miss line_size="64" mem_area_id="2">
        <distribution>
          <uniform min="1000" max="2000"/>
        </distribution>
      </i_miss>
      <d_miss line_size="64" mem_area_id="3">
        <distribution>
          <normal mean="200" standard_deviation="30"/>
        </distribution>
      </d_miss>
    </cache>
  </resource>
</platform>
```

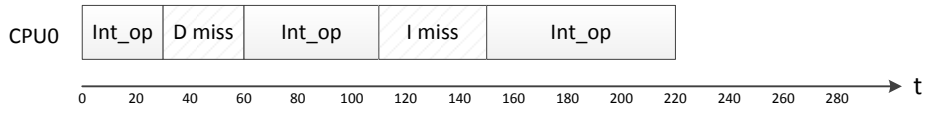

Figure 5.6: Example of a cache definition for processing element. Distributions used in the definition randomly chooses the amount of clock cycles to the next cache miss. That amount is generated again every time cache miss occurs. Timeline shows a task executing 160 integer operations on a PE `cpu0`. The execution is stalled for 30 cycles due to data cache miss and 40 cycles for instruction cache miss, and thus taking 230 cycles to complete.

Tags `i_cache` and `d_cache` define the cycle count between cache misses for instruction and data memories with three properties.

Line size defines the amount of bytes read from the memory every time the corresponding cache miss occurs.

Memory area links the cache miss to be read from a certain memory region.

Distribution is used to randomly select the number of execution cycles to the next cache miss.

Cache miss definitions don't need port definitions as they are defined only for the PE and the port information is only needed to define task and trigger specific sources and destinations.

6. MEMORY MODEL

TG's memory model consists of DRAM models and memory areas mapped to them. The DRAM memories for the Transaction Generator are modeled with the help of Accurate DRAM Model (ADM) package [44], which have been integrated to TG as a part of this thesis.

6.1 Accurate DRAM Models

ADM is a SystemC package implemented with OCP-IP TLM sockets that models dynamic random-access memories accurately on transaction level. ADM is developed by Royal Institute of Technology (KTH) with OCP-IP's Network on Chip Benchmarking working group. Package models the important aspects of DRAMs that greatly affects the memory access times, such as data rates, refresh rates and delays between accesses. Figure 6.1 shows an example of a DDR1 memory configuration.

```
# 256MB X8 DDR1 DRAM, JEDEC DDR RAM at 200MHz

clockPeriod i:5          # IO clock period
dataRate i:1             # DRAM data rate, 1 = "DDR1"
refreshPeriod i:7600     # DRAM refresh period
refreshDuration i:120    # DRAM refresh duration
addressBusWidth i:25     # bit width of the address bus
bankAddressWidth i:2     # bit width of bank address
rowAddressWidth i:13     # bit width of row address
columnAddressWidth i:10  # bit width of column address
dataWidth i:4            # DRAM IO width
burstLength i:4          # DRAM Minimum burst length
tHopRow i:55             # Delay caused by row hops
tRCD i:15                # Delay, Activate to RD/WR
tCAS i:10                # Delay, Read to first response
tDQSS i:10               # Delay, Write to first data registered
tWTR i:10                # Delay, Read to last write registered
```

Figure 6.1: Example of a ADM memory configuration file for a 256MB X8 DDR1 memory.

Configuration file describes, for example, the clock cycle period (5 ns = 200 MHz), various address widths in bits, the duration of refresh and the rate it occurs (120 ns long refresh every 7600 ns), and several timing parameters.

6.2 Memory Areas

TG's memory model is divided into memory areas that model the contiguous memory regions. They represent the data and instruction memory regions accessed by the

tasks and PEs. Figure 6.2 visualizes memory address space divided to two DRAM memories and three memory areas mapped to those.

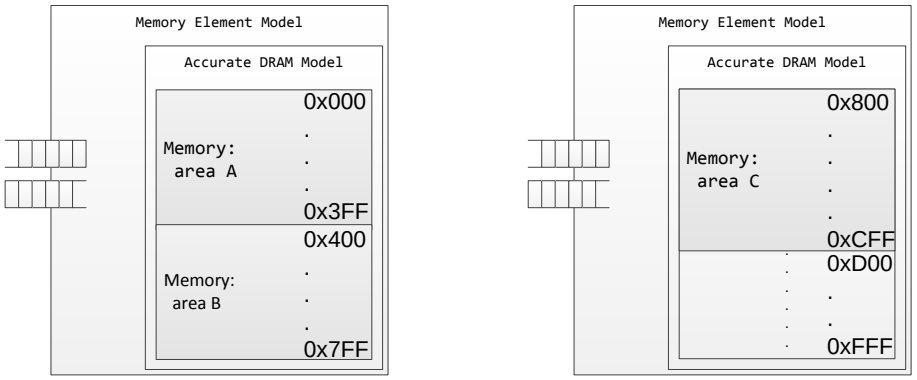

Figure 6.2: Example of the contiguous address space divided to two memory models and the memory areas mapped to them.

When TG's task model generates a send or read transaction to a memory model it is always assigned to a certain memory area. Both the reads and sends are defined to happen in bursts. The burst size defines how many of the bytes being read or written are on consecutive memory addresses. When a task writes a token, that is bigger than the burst size, it is split into multiple pieces when it arrives to the memory model. For each of the pieces a new random address within the corresponding memory area is generated and thus the writes might, for example, trigger memory line changes that affect the operations execution speed.

Similarly the burst size is defined for memory reads. If the read request defines a larger token than the burst size to be read from the memory the memory is read from multiple random places within the corresponding memory area.

ADM package models only the memories and thus there is a simple wrapper around the memory to model its network interface that connects it to the NoC model. Figure 6.3 shows the XML definition of a memory model for TG.

The ADM package has its own configuration files to describe the details of the memory being modeled. For TG the memory model is defined with following parameters.

Frequency specifies the operating frequency of the network interface. It doesn't need to be related to the speed of the memory itself.

RX buffer size can be used to limit amount of incoming writes and read requests that are not yet processed by the memory. It models a simple FIFO queue between the memory and the network.

```
<platform>
  <resource id="1" name="DDR1"
            frequency="200" class="memory"
            rx_buffer_size="512" tx_buffer_size="64"
            packet_size="16" request_size="8"
            ocp_param="examples/ocpParams"
            adm_param="examples/admParams">
  </resource>
</platform>
```

Figure 6.3: Example of a memory definition for the platform. Memory has a network inteface operating at 200 MHz that has 512-byte receive and 64-byte send buffers. It sends the read data in 16-byte packets and the read request needs 8 bytes. The details of the DRAM model is configured in *ocpParams* and *admParams* files.

TX buffer size similarly limits the amount of data read from the memory that is not yet sent to the network.

Packet size states the maximum size of a single network data packet that the network interface can send to the network. If bigger chunks of data are read from the memory they are automatically split to this size and reconstructed at the receiving network interface.

Request size describes the size of a read request for the memory. As TG supports only split-transaction NoC models every time a task issues a read it first sends a read request to the memory model and waits for the memory to send back the requested data.

7. NETWORK MODEL

The Network-on-Chip (NoC) models are not a part of TG itself. TG uses a simple queue-interface to the NoC models allowing various NoC implementations to be easily attached to it. The NoC to be used in the simulation is defined in the simulation constraints section and can be easily selected without modifying the application or platform models. The tag has three predefined attributes `class`, `type` and `subtype` that are used to select a specific NoC implementation to construct for the simulation, for example a 2x2 mesh implemented with OCP-IP TLM sockets, as in figure 7.1. All the other tags inside the definition are also parsed by TG and passed to the constructor. Figure 7.1 shows the NoC description in the XML source for `fh_mesh_2d`, one of the example networks provided with TG.

```
<system>
  .
  .
  <constraints>
    .
    .
    <noc class="fh_mesh_2d" type="ocptlm" subtype="2x2"
         pkt_switch_en_g="1"
         stfwd_en_g="0"
         addr_width_g="32"
         packet_length_g="8"
         timeout_g="5"
         fill_packet_g="0"
         len_flit_en_g="1"
         oaddr_flit_en_g="0"
         fifo_depth_g="4"
         noc_freq_g="50000000"
         ip_freq_g="50000000"/>
  </constraints>
</system>
```

Figure 7.1: Example of a NoC definition used to select `fh_mesh_2d`, one of the example interconnection networks, to use in the simulation. The tags after subtype are also passed to the network constructor to define its implementation-specific parameters, such as operating frequencies, buffer depths and network interface details.

NoC models to use with TG can be described in C, C++ or SystemC for platform-independent simulation or in hardware description languages, such as VHDL and Verilog. For mixed language simulations TG can be compiled and simulated, for example with Modelsim [28]. Network models can be described in RTL or in higher abstraction levels, such as TLM, which provide speedup for the simulation time as TG has also been implemented at TL.

7.1 Custom NoC Integration

Custom NoC models can be added to TG by adding a constructor to TG's NoC library. The NoC library has a tree of factories that get called in the beginning of the simulation to select a correct NoC constructor based on the `class`, `type` and `subtype` attributes. Figure 7.2 depicts TG's NoC Factory structure.

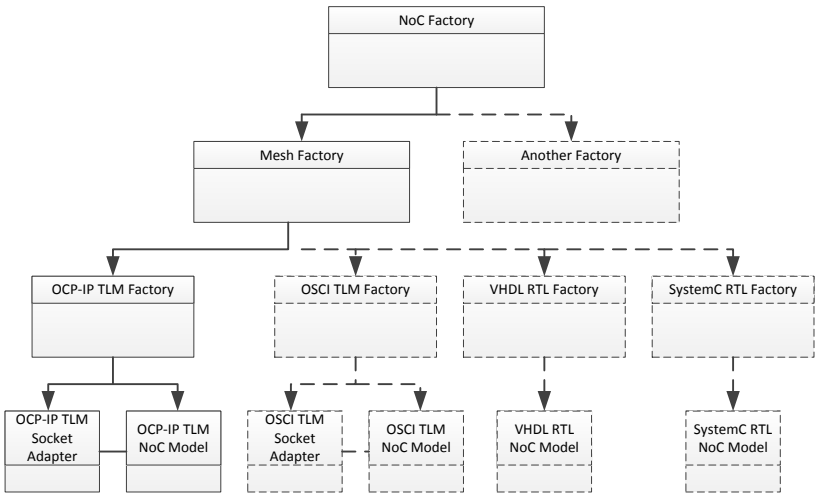

Figure 7.2: Figure illustrates TG's factory tree. To add a custom NoC to simulate with TG, user must add a constructor for it to the NoC Factory class.

Figure 7.2 illustrates the selection of the network model for simulation. The topmost NoC Factory class includes the header files of all network models and contains a pointer to each one. Based on the `class` attribute one of them is constructed with a NoC Configuration Interface object that carries the parameters from the XML source and means for the network model to communicate with TG.

For the user it is enough to modify the general NoC Factory class, but the example networks have a deeper tree structure. The Mesh Factory class, for example, has been divided into multiple smaller factory classes depending on their implementation language and technology. One of them is instantiated based on the `type` attribute. This is to make the recompilation time shorter, if one of the lowest level classes is modified, and cleaner inclusion of the Hardware Description Language (HDL) network implementations, which can't be compiled for TG without a compiler capable of mixed-language simulation.

The last layer chooses, based on attribute `subtype`, the actual network implementation, such as the 2-by-2 mesh in figure 7.1. In the example network models there's for most of them different template based implementation classes, mainly because SystemC taking port widths as a template parameter preventing them to be created easily dynamically.

TG has a native simple interface for communicating with RTL models and adapters to communicate with TLM models, as in the example of figure 7.2. The adapters implement a conversion of TG native interface to SystemC and OCP-IP TLM sockets. With the native method network models are connected to TG's PEs and memories throug a simple queue interface, which is provided for the network model's constructor. The queue interfaces provide the following functionality for the network models to communicate with TG.

rxPutPacket method to pass the data towards TG as it is received from the network.

rxSpaceLeft method which informs the amount of bytes left in the receive buffer. If the buffer doesn't have enough space for the incoming packet it must not be passed to the receiver thus stalling the reception.

rxBufSize method returns the total size of the receive buffer.

rxGetReadEvent method returns SystemC event that is activated every time the receive buffer has been read by the platform model. This can be useful for higher abstraction level models preventing the need to poll the interface.

txPacketAvailable method can be used to poll the resource whether it has a data packet to send.

txGetPacket method returns the raw data to send through the network as well as information about the transmission for the network to route it to the receiver.

txGetPacketAvailableEvent method returns a SystemC event that is activated every time the resource has a data packet to send.

TG uses a similar interface for the PEs and memories from the other of the buffers to model the interface to the NoC. This interface is provided to simplify connecting the NoC as the buffer models are more unnecessarily complex for the NoC model on the TG's side.

7.2 Provided NoC Example Models

TG provides example NoC implementations, such as simple bus, crossbar, ring and two 2-D meshes. Example networks are released also as a part of Funbase IP library [34]. TG release packet includes example NoCs described in RTL and TLM, from which the VHDL versions, with the exception of `ase_mes`, are described in [40]. The SystemC version were implemented for the author's bachelor thesis [23] and their effect on speedup with TG presented in [25].

Modeling on higher abstraction levels is beneficial, simulation speed wise, for TG as it is also modeled on higher abstraction level. With higher abstraction level modeling many of the implementation details can be left out offering faster simulations with TG. Comparing the SystemC RTL and TLM modeling styles for TG, a 10-40x speedup in simulation time can be achieved easily without loosing too much accuracy, as reported in [25].

Figure 7.3 illustrates the difference between RTL and TLM models provided with the package. RTL models are simulated every time some signal changes, that is about 30 times in the example. In contrast, TLM models are simulated only at the beginning and the end of transactions, the 4 phases in the example. However, the exact timing cannot always be determined in TLM leading to estimation errors.

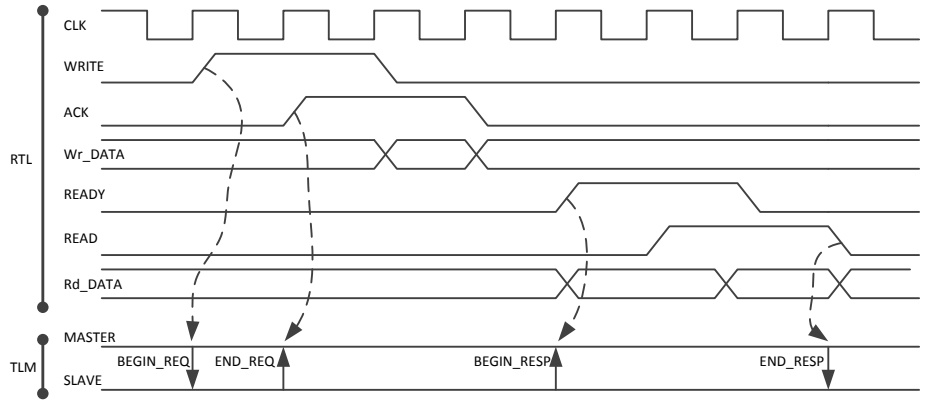

Figure 7.3: Conceptual difference between communication of RTL and TLM models. While RTL models describe every signal changes at clock cycle detail, in TLM the communication is described only annotating the important moments, such as the beginnings and ends of requests and responses.

Table 7.1 lists the example network-on-chip models that are distributed with TG.

TG's major idea is compare different NoC implementations. Thus, the user is expected to provide their own NoC implementations. The example NoC models are provided mainly to offer examples of how different implementation can be integrated to TG.

Table 7.1: Table listing the example NoC implementations provided with TG.

Name	Language	Explanation
Simple Bus	SystemC OCP-IP TLM	Shared multimaster bus without detailed timing information. It gives an example of a SystemC OCP-IP TLM socket based network integration with TG's socket adapters.
FH Crossbar	VHDL	A high throughput interconnection. Impractical for larger systems in real life, due to large number of long wires, but for simulation it offers a way to get performance estimations for near-optimal interconnections.
FH Ring	VHDL	A simple 1-D multi-hop topology connecting resources as a two-way ring.
FH 2-D Mesh	VHDL SystemC RTL SystemC TLM SystemC OCP-IP TLM	Mesh implementation with configuration options, for example to the packet structure, switching and buffering.
Ase 2-D Mesh	VHDL	Minimal implementation of the 2-D mesh topology optimized for small size and short delays in routing

8. SUMMARY OF TG

This chapter provides overview of what was implemented or modified for this thesis and summary of TG's current implementation package.

8.1 New Features

TG implementation was completely refactored from TCL and SystemC 1 based code generator to self-contained simulation core with up to date technologies, such as SystemC 2 TLM and OCP-IP TLM sockets. One major addition was the memory area models and DRAM memory models implemented with the ADM package.

Support to convert and simulate MCSL NoC Traffic Patterns was integrated as a part of TG. Modeling of task workload was diversified by implementing distributions to calculate the workload, and operations to read from and write to memories. PE models were enhanced with inclusion of a new scheduling algorithm, cache model and possibility to have finite buffers for token reception and transfers. Measurements of the simulation were made more detailed and new measurements were added, such as measuring token's traverse time from one place to another.

8.2 Implementation

TG is implemented in C++ with the SystemC library handling the notion of time and concurrency. It uses few libraries from Boost [8], such as Asio to handle TCP connection to Execution Monitor, Program Options for command line parsing, and Property Tree for parsing and storing the XML input models.

The implementation is divided into 15 classes, which mainly follow the tag structure of the input XML model descriptions. Main reason for the current class division was to handle parsing the input XML tags in smaller pieces, so that every class handles only its own information and the sub-tags would be parsed in their own respective classes. Figure 8.1 presents the class diagram of the current implementation.

Configuration class parses the constraints from the model description and holds the general information of the simulation, such as the simulation's length and time resolution, and the mapping information, for example which task is mapped where.

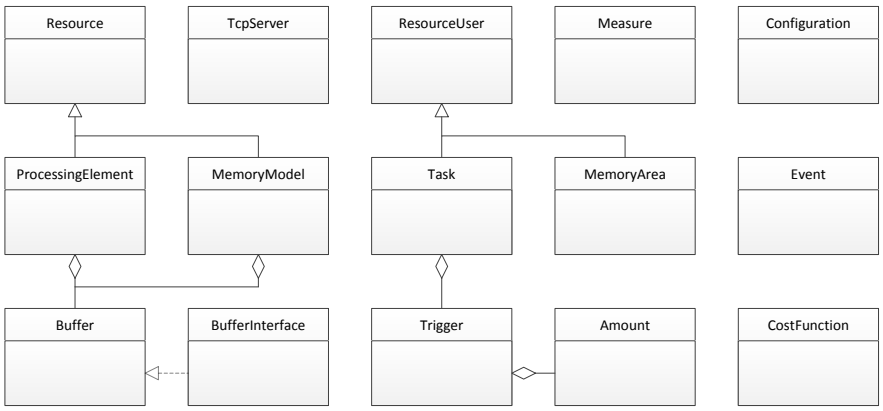

Figure 8.1: Class diagram of TG's current implementation showing only the main relationships between classes.

Amount class parses the polynomials and distributions defined, for example, for the computation operations and communication's byte amounts. During the simulation this class allows the evaluation of the polynomial and distributions as defined in the model description.

CostFunction class parses the cost functions and implements a simple calculator to evaluate them at the end of simulations. With the cost functions the user can, for example, compare the various NoC implementations and the effect of its configuration parameters.

Event class parses the event descriptions and implements a SystemC thread to fire the events during the simulation.

Buffer class implements the PE's internal memory model and the communication interface between the TG's resource models and the network model.

BufferInterface class defines the interface that is exposed to the network model from the Buffer classes.

Measure class implements SystemC threads to handle the measurements gathered during the simulation and the communication with Execution Monitor.

Resource class acts as a base class for the MemoryModel and ProcessingElement. It parses the information that is common between them, such as operating frequency, buffer sizes, and packet size.

MemArea class parses the information related to memory areas.

MemoryModel class parses the information related to memory elements and handles the communication with the ADM DRAM models.

ProcessingElement class parses the information related to PEs and implements features, such as cache miss models, task scheduling algorithms, the execution of tasks and their communication. It is the most important class implementing majority of TG's MoC.

ResourceUser class is a base class for Task and MemArea classes handling the parsing of their common information, such as identification numbers, and input and output ports.

Task class parses the general task information from the task tags and implements task model's internal state machine.

Trigger class parses the trigger tags and handles the list of operations to execute after being fired.

TcpServer class constructs a TCP server for communication with Execution Monitor.

TG package includes both example and tutorial application models, application models described in [37], and the MCSL NoC Traffic Patters [26]. In addition to TG simulation core with the ADM package [44], it comes with SystemC and VHDL NoC models and the Execution Monitor [17]. Table 8.1 lists the current source code size of TG.

Table 8.1: Table showing the size of the TG codebase. Application models include the application workload models, examples and tutorials provided with TG. C++ files are divided to TG simulation core, the ADM package and NoC models imlemented in SystemC. Package also provides VHDL NoC implementations and the Execution Monitor program.

Language	Part	Files	Comment Lines	Code Lines
XML	Application models	95	1 303	22 618
C++	TG core	42	2 517	7 532
	ADM	12	352	1 332
	NoC models	75	3 707	12 054
	Total	129	6 576	20 918
VHDL	NoC models	53	2 883	10 541
Java	Execution Monitor	59	4 166	9 436

Implementation process of the new version of Transaction Generator was carried out without many complications. One complication arose with the SystemC library's choice of implementing wire's width information as a template parameter.

Templates in C++ require the information to be known at compilation time, thus defeating the TG's idea of dynamic construction of the models without needing a recompilation. At the moment this issue has not been solved for the instantiation of the example RTL network models. TG doesn't use internally any templates for the parts that depend on the application model and thus doesn't need recompilation between simulations.

Another complication was caused by the Modelsim and its compiler for SystemC. The compiler couldn't handle, at least at the time when the refactoring of TG started, many nice-to-have C++ features, such as smart pointers, leading to a more C-like implementation of TG and the small compilation differences when compiled to Modelsim and native program. For example, with the Modelsim the TCP server is not included in the compilation and the command line parameters are parsed differently.

9. CASE STUDY

This chapter provides examples of TG used to examine MCSL's H.264 video decoder application model for 720p resolution. Application model is first simulated with two different NoCs of 2-D mesh topology. Then the application simulated with the faster NoC with varying packet sizes. Lastly the effect of PE mapping to the NoC is examined. The NoC's chosen for the example simulations are `fh_mesh_2d` and `ase_mesh`. The point of this chapter is to show few examples of the NoC analysis that can easily be carried out in a single work day.

9.1 H.264 Application Model

The MCSL NoC Traffic Patter application model for H.264 video decoder with 720p resolution recorded traffic pattern [26] is used in these examples. Figure 9.1 shows a snippet of the application model. The MCSL application model is converted to TG's native format to allow easier modification of the simulation parameters, such as the used NoC model, packet sizes for the NoC, and NoC operating frequency. Application model consists of 50 tasks with 73 communication connections running on 11 PEs. PEs are mapped to 4x4 2-D mesh NoC leaving 4 of the possible PE or other resource connections empty. Application sends around 200-byte packets between the tasks and the operation counts are in the range of 2000-4000 per trigger.

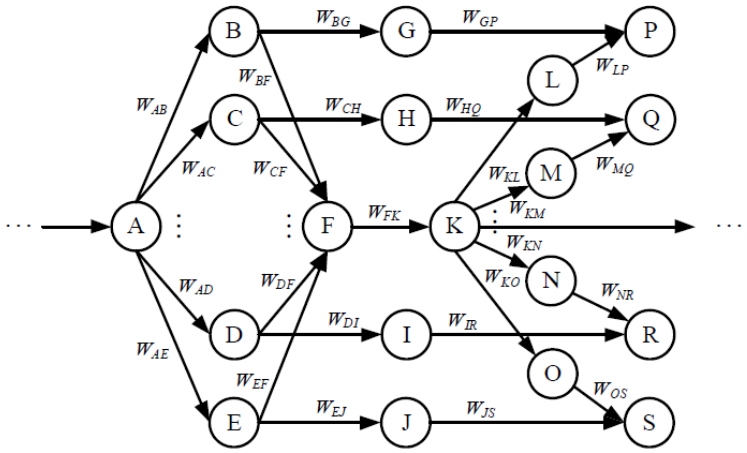

Figure 9.1: A part of the H.264 application model [26]. Circles with letters correspond to tasks and edges to the communication with weights as the size of transferred data.

In the experiments the PEs are operating at a constant frequency while the NoC frequency is varied in a range of 10-20% of the PE frequency. Effect of four network packet sizes and three mappings are examined.

9.2 Difference Between NoC Implementations

First set of simulations show the difference in execution time of the application between `fh_mesh_2d` and `ase_mesh` NoC models. The application is simulated 20 times to get more realistic average execution time, for example by avoiding the effect of empty network at the beginning of simulation. The network models used in this example are both using the same network interface implementation, which handles, for example the clock domain crossing with asynchronous FIFOs. Difference in the application's execution time is thus caused only by the router and link implementation divergence.

Both of the NoCs are using fixed XY routing for at maximum of 16-byte network packets. Both use wormhole switching and fixed arbitration when selecting the next packet to transfer. The main difference between the NoCs affecting the speed in clock cycles is that the `fh_mesh_2d` has a FIFO buffers on every link and uses 1 or 2 clock cycles between network packets to switch the next packet while the `ase_mesh` NoC has no extra buffering on links and is capable of switching packets to links without any idle clock cycles between the packets. Figure 9.2 shows the simulation results for both NoCs. Results are normalized to the faster NoC simulated with the slowest operation frequency. Application's execution speed is measured from the moment all of the tasks have been executed 20 times, as is the case in all the simulation results shown in this chapter.

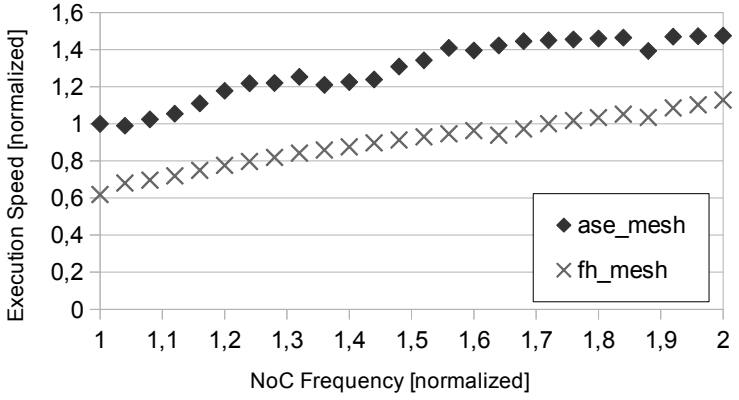

Figure 9.2: Simulation speed comparison between `fh_mesh_2d` and `ase_mesh` as a function of NoC frequency. Speed is normalized to the application's execution time with `ase_mesh` on it's slowest operating frequency in the simulation. Frequencies are increased up to two times faster frequency.

From the simulation results presented in figure 9.2 a clear performance difference (around 1.45x) can be seen between the NoCs. Even though the throughput is equivalent for the networks, the shorter latency of `ase_mesh` leads to significant speedup for the application. Result of doubling the NoC operation frequency was about 1.4x. Also the nonlinear relationship for the increase of the NoC's operating frequency can be seen. Even a slight difference in the operating frequency can cause the traffic congestion to build up differently especially for the FIFO-less `ase_mesh` and cause noticeable slowdown even when the operation frequency is increased.

9.3 Influence of Network Packet Size

Individual network parameter configuration can make a great difference for the performance of the application. NoC `ase_mesh` is selected to more precise examination for the application, as it was significantly faster in the previous measurements. The `ase_mesh` implementation is realized with minimal features and its behavior can't be configured. Nevertheless, it can behave quite differently based on the size of the packets sent through it, even if the size of the actual payload is the same. Figure 9.2 shows the simulation results for different network packet sizes as a function of NoC operating frequency.

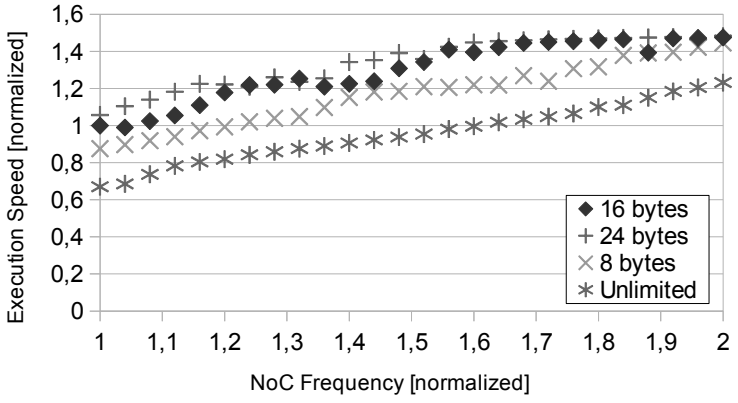

Figure 9.3: Application's execution speed for various network packet sizes as a function of NoC operating frequency.

The NoC implementation doesn't impose any restrictions on the network packet size. Every packet is prefixed with a one word network address field followed by the payload. The network packet can't be split on its way in the network and if it is long it reserves the links for a long time.

As seen in figure 9.3 packet size affects the performance of the application significantly when using the `ase_mesh`. The unlimited packet size results in about 200-byte packets being sent through the network blocking the links for long periods

for one packet and leading to slower execution. Smaller packet size increases the communication overhead as every packet needs its own network header, which, for example leads to 33% traffic increase for the 8-byte packets. For smaller NoC operation frequencies the 24-byte packet is faster but the speedup to 16-byte packets gets smaller as the frequency is increased.

9.4 Effect of PE Mapping

How the application is mapped to the PEs and the PEs to the NoC can affect the application's performance significantly. Mapping has been discussed thoroughly, for example in [35]. Possible mapping variations grow exponentially for the amount of processing elements and the tasks. This leads to the impracticality of evaluating all of them and in this example only few variations is shown as an example. Figure 9.4 reports the effect on application's execution speed by simple mapping variations of rotating the original PE mapping by 90 and 180 degrees. This affects the execution speed especially when using the `ase_mesh` NoC as it has fixed routing and arbitration schemes.

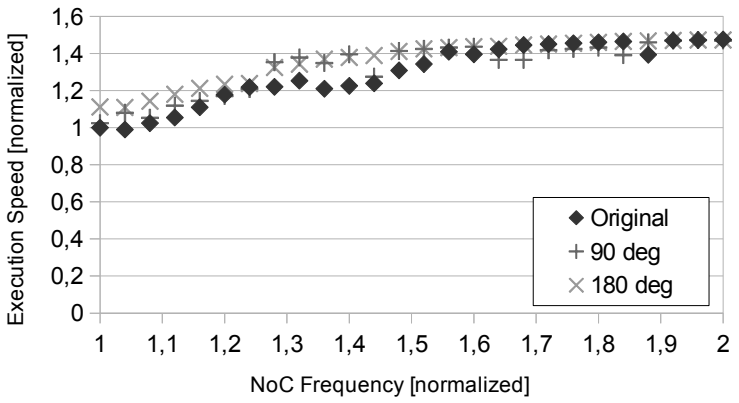

Figure 9.4: Application's execution speed for different PE mappings to the NoC as a function of NoC operating frequency. Original depicts the results for the mapping the application model comes with and 90 deg and 180 deg the result for the original mapping rotated by 90 and 180 degrees respectively.

Mapping has consequences on many aspects of the application, such as PE utilization, communication latencies, and congestion on the network, that can be monitored with TG.

10. CONCLUSIONS

Transaction Generator was re-implemented with up-to-date technologies with the C++ programming language using SystemC version 2 library to handle the notion of time and concurrency needed for describing hardware. Additional libraries for the implementation used were the OCP-IP TLM kit to support OCP-IP TLM sockets and Boost to help with trivial tasks, such as XML parsing.

Memory models were added with the help of the DRAM models from the ADM package. Inclusion of DRAM models allowed the implementation of reading and writing operations for the tasks, and the simple cache miss model for the processing elements. Support to simulate and convert the MCSL NoC Traffic Patterns application models were added bringing the supported application model set a healthy addition.

Three example simulation cases were evaluated for the H.264 video decoder application model providing information of the effect of the difference between two different NoC implementations, the network packet size, and different PE mappings to the network. The results, showing a difference of around 0.6-1.45x speedup for the application execution speed, provide an example of $\sim$100 simulations that can be easily executed and analyzed within a work day with TG.

Also a detailed description of the current state of the simulation models of Transaction Generator was provided. The result of this thesis, consisting of around 10 000 lines of code for the simulator's core, has been used in research, for example in [40, 25] and [37], and is currently available from Accellera [45]. The objective for this thesis has been achieved. Table 10.1 shows a summary of TG.

For future work, Transaction Generator's simulation logging should be expanded and refined with more details to allow the user to analyze easier the actual reasons why something behaves like it does in the simulations. For example, finding the root cause of the bottlenecks and the reason for unexpected or surprising results could be made better. For example, the slow-downs in the example simulations, when the operating frequency was increased, can be explained in general but the actual tasks causing it are harder to find.

More thorough analysis of the application models would also benefit the user, for example to make more obvious selecting the models that would be better applicable for a certain application the user is benchmarking the NoC for. Also, even though

out of the TG's scope, the user would benefit from having a possibility of simulating workload models generated automatically from existing legacy code.

Table 10.1: Table of the TG features.

Category	Details
Purpose	NoC/MP-SoC simulation and analysis.
Model style	Workload model for NoC.
Application model	KPN based or MCSL traffic pattern.
Supported NoC styles	From RTL to more abstract (TLM).
Supported NoC descriptions	SystemC, VHDL, Verilog...
Provided test cases	10 (native) + 8 (MCSL).
Provided example NoCs	5.
Language	C++ with SystemC.
Needed libraries	OCP-IP TLM Kit, Boost.
Code lines	∼10k (core).
License	LGPL.
Available	NoCBench [31], OCP-IP [33], Accellera [45].

REFERENCES

[1] P. Abad, P. Prieto, L. G. Menezo, A. Colazo, V. Puente, J.-A. Gregorio, *TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers*, Sixth IEEE/ACM International Symposium on Network on Chip (NoCS), Pages 99- 106, May 2012.

[2] Academy of Finland, *Academy of Finland*, `http://www.aka.fi/en-GB/A/`, Referenced 19.4.2014.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone and A. Sangiovanni-Vincentelli, *Metropolis: an integrated electronic system design environment*, Computer, Volume 36, Number 4, Pages 45- 52, 2003.

[4] L. Benini, G. De Micheli *Networks on chips: a new SoC paradigm*, Computer, Volume 35, Number 1, Pages 70- 78, January 2002.

[5] Y. Ben-Itzhak, E. Zahavi, I. Cidon, A. Kolodny, *HNOCS: Modular open-source simulator for Heterogeneous NoCs*, International Conference on Embedded Computer Systems (SAMOS), Pages 51- 57, July 2012.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Waish, M. D. Hill, D. A. Wood, *The GEM5 Simulator*, SIGARCH Computer Architecture News, Volume 39, Number 2, May 2011.

[7] T. Bjerregaard and A. Mahadevan, *A survey of research and practices of Network-on-chip*, Journal ACM Comput. Surv., Volume 38, Issue 1, ISSN 0360-0300, ACM, New York, NY, USA, June 2006.

[8] Boost, *Boost C++ Libraries*, `http://boost.org`, Referenced 6.5.2014.

[9] L. Cai, D. Gajski, *Transaction Level Modeling: An Overview*, International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS), Pages 19-24, October 2003.

[10] A.S. Cassidy, J.M. Paul and D.E. Thomas, *Layered, multi-threaded, high-level performance design*, Design, Automation and Test in Europe Conference and Exhibition, Pages 954- 959, 2003.

[11] W. Dally, and B. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers Inc., ISBN 0122007514, January 2004.

[12] W. J. Dally, B Towles, *Route packets, not wires: on-chip interconnection networks*, Design Automation Conference (DAC), Pages 684- 689, 2001.

[13] M. Gries, *Methods for Evaluating and Covering the Design Space during Early Design Development*, Integration, the VLSI Journal, Volume 38, Pages 131- 183, 2003

[14] K. Holma, M. Setälä, E. Salminen, M. Hännikäinen and T. D. Hämäläinen, *Evaluating the Model Accuracy in Automated Design Space Exploration*, Microprosessors & Microsystems: Special Issue in Dependability and Testing of Modern Digital Systems, Volume 32, Issue 5-6, Pages 321- 329, April 2008.

[15] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips (Systems on Silicon)*, Morgan Kaufmann Edition 1, ISBN 012385251X, October 2004.

[16] N. Jiang, D. U. Becker, G. Michelogiannakis, B. Towles, D. E. Shaw, J. Kim, W. J. Dally, *A detailed and flexible cycle-accurate Network-on-Chip simulator*, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Pages 86- 96, April 2013.

[17] K. Holma, T. Arpinen, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, *Real-time execution monitoring on multi-processor system-on-chip*, International Symposium on System-on-Chip (SOC), Pages 1- 6, November 2008.

[18] G. Kahn, *Natural Semantics*, Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), Springer-Verlag, Lecture Notes in Computer Science, Volume 247, Pages 22-39, 1987.

[19] T. Kangas, J. Riihimaki, E. Salminen, K. Kuusilinna, T. D. Hämäläinen, *Using a communication generator in SoC architecture exploration*, International Symposium on System-on-Chip, Pages 105- 108, November 2003.

[20] K. Keutzer, A. R. Newton, J. M. Rabaey, A. Sangiovanni-Vincentelli, *System-level Design: Orthogonalization of Concerns and Platform-based Design*, Trans. Comp.-Aided Des Integ. Cir. Sys., Volume 19, Number 12, Pages 1523- 1543, November 2006.

[21] T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, D. Bussaglia and M. Ariyamparambath, *Virtual architecture mapping: A SystemC based methodology for architectural exploration of system-on-chip designs*, Proc. of the Int. workshop on Systems, Architectures, Modeling and Simulation (SAMOS), Pages 138- 148, 2003.

[22] K. Lahiri, A. Raghunathan, and S. Dey, *System-level performance analysis for designing on-chip communication architectures*, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Volume 20, Number 6, Pages 768-783, 2001.

[23] L. Lehtonen, *Mikropiirinsisäisten kytkentäverkkojen mallintaminen SystemC-kielellä*, Bachelor Thesis, Tampere Univesity of Technology, 22 pages, 2012.

[24] L. Lehtonen, E. Pekkarinen. *Transaction Generator Technical*, `http://www.tkt.cs.tut.fi/research/nocbench/data/sctg2_technical.pdf`, Retrieved 12.1.2012.

[25] L. Lehtonen, E. Salminen and T. D. Hämäläinen, *Analysis of Modeling Styles on Network-on-Chip Simulation*, Norchip Conference, Tampere, Finland, November 2010.

[26] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast and Z. Wang, *A NoC Traffic Suite Based on Real Applications*, 2011 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Pages 66- 71, ISSN 2159-3469, July 2011.

[27] A. Mello, A. Amory, N. Calazans, F. Moraes, *ATLAS - A NoC Genration and Evaluation Framework*, Design, Automation & Test in Europe DATE, 2011.

[28] Mentor Graphics, *Modelsim*, `http://www.mentor.com/products/fpga/model`, Referenced 22.4.2014.

[29] A. Moonen, M. Bekooij, R. van den Berg and J. van Meerbergen, *Evaluation of the throughput computed with a dataow model - A case study*, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems, ISSN 1574-9517, 2007.

[30] G.E. Moore, *Cramming More Components Onto Integrated Circuits* Proceedings of the IEEE, Volume 86, Number 1, Pages 82- 85, ISSN 0018-9219, January 1998.

[31] NOCBENCH Project, *Standardization of Benchmarking Methodology for Network-on-Chip*, `http://www.tkt.cs.tut.fi/research/nocbench`, Referenced 19.4.2014.

[32] Noxim, University of Catania, *Noxim*, `http://www.noxim.org`, Referenced 18.4.2014.

[33] OCP-IP, *Open Core Protocol - International Partnership*, `http//www.ocpip.org`, Referenced 12.1.2012.

[34] OpenCores, *Funbase IP library*, `http://opencores.org/project,funbase_ip_library`, Referenced 23.4.2014.

[35] H. Orsila, *Optimizing Algorithms for Task Graph Mapping on Multiprocessor System on Chip*, PhD Thesis, Tampere University of Technology, Publication 972, 199 pages, 2011.

[36] E. Pekkarinen, *Moniprosessorisovellusten mallintaminen*, Bachelor Thesis, Tampere University of Technology, 26 pages, 2011.

[37] E. Pekkarinen, L. Lehtonen, E. Salminen, T. D. Hämäläinen, *A Set of Traffic Models for Network-on-Chip Benchmarking* International Symposium on System-on-Chip, Pages 78- 81, October 2011.

[38] A.D. Pimentel, L.O. Hertzbetger, P. Lieverse, P. van der Wolf and E.E. Deprettere, *Exploring embedded-systems architectures with Artemis*, Computer, Volume 34, Number 11, Pages 57- 63, ISSN 0018-9162, November 2001.

[39] Qualcomm, *Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age*, Qualcomm white paper, October 2011, `https://developer.qualcomm.com/download/qusnapdragons4whitepaperfnlrev6.pdf`, Retrieved 18 April 2014.

[40] E. Salminen, *On Design and Comparison of On-Chip Networks*, PhD Thesis, Tampere University of Technology, Publication 872, 230 pages, 2010.

[41] E. Salminen, C. Grecu, T. D. Hämäläinen and A. Ivanov, *Application modeling and hardware description for network-on-chip benchmarking*, Computers Digital Techniques, IET, Volume 3, Number 5, Pages 539- 550, September 2009.

[42] E. Salminen, A. Kulmala and T. D. Hämäläinen, *Survey of Network-on-chip Proposals*, OCP-IP, `http://www.ocpip.org/uploads/documents/OCP-IP_Survey_of_NoC_Proposals_White_Paper_April_2008.pdf`, Retrieved 22.1.2012.

[43] A. Sangiovanni-Vincentelli, *Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design*, Proceedings of the IEEE, Volume 95, Number 3, Pages 467- 506, ISSN 0018-9219, March 2007.

[44] K. Srinivasan, E. Salminen, *A memory Subsystem Model for Evaluating Network-on-Chip Performance*, OCP-IP white paper, September 2010.

[45] SystemC, *Open SystemC Initiative, OSCI*, `http://www.accellera.org/home`, Referenced 12.1.2012.

[46] A. Varga et al., *The OMNet++ discrete event simulation system*, European Simulation Multiconference (ESM), Pages 319- 324, June 2001.

[47] W. Wolf, A. Jerraya and G. Martin, *Multiprocessor System-on-Chip (MPSoC) Technology* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN 0278-0070, Volume 27, Number 10, Pages 1701- 1713, October 2008.